우리식조작체계문고

Linux핵심부 해설서

교육성교육정보쎈터
주체97(2008)

차 례

머 리 말

위대한 령도자 **김정일**동지께서는 다음과 같이 지적하시였다.

《전자공학부문의 과학자, 기술자들은 이미 이룩한 성과와 경험에 기초하여 전자공학과 전자공업을 새로운 높은 단계에로 발전시키며 인민경제 중요부문의 전자계산기화, 로보트화를 실현하기 위한 투쟁을 힘있게 벌려야 합니다. 새로 개발한 극소형계산기에 쓰이는 전자요소와 전자재료들의 특성을 개선하고 자급률을 높이며 프로그람을 적극 개발하고 전자계산기의 리용분야를 넓혀나가도록 하여야 합니다.》(《**김정일**선집》 제12권, 200페지)

오늘날 과학과 기술이 급속히 발전하고 생산과 경영활동에서 부문별 련계가 보다 밀접해지고있는 사정은 정보기술을 하루빨리 발전시켜 인민경제 모든 부문에서 콤퓨터화수준을 보다 높은 수준에로 끌어올릴것을 요구하고있다.

특히 세계적으로 정보산업분야에서 기술경쟁이 그 어느 분야에서보다 치렬하게 벌어지고있는 사정은 우리 식의 프로그람을 적극 개발하여 나라의 정보산업을 최단기간내 세계적수준에 끌어올릴것을 절박하게 요구하고있다.

위대한 령도자 **김정일**동지의 현명한 령도밑에 우리 나라의 과학자, 기술자들은 정보산업시대의 요구에 맞게 우리 식의 조작체계 《붉은별》을 개발함으로써 주체적인 정보기술발전에서 획기적인 전환을 가져오게 하였다.

이 책에서는 조작체계에서 가장 기본적인 프로그람부분인 핵심부프로그람에 대하여 서술하였다. 구체적으로 원천코드가 완전히 공개된 Linux핵심부 2.6.9판을 대상으로 그 동작원리와 프로그람적인 실현에 대하여 8개장에 걸쳐 해설하였다.

우리는 프로그람기술에 정통하고 새로운 프로그람들을 적극 개발함으로써 사회주의강성대국건설에 크게 이바지하여야 한다.

제 1 장. 핵심부의 기초개념

제 1 절. 기본개념

1. Linux의 발전력사

Linux는 Unix계조작체계로서 그 발전을 Unix의 발전과 떼여놓고 생각할수 없다.

1) Unix의 발전

Unix체계는 근 40년의 력사를 가지고있다. 첫 Unix체계는 1969년 벨연구소가 개발하였다. 이 체계는 완전히 C언어로 되여있으며 아쎔블러어가 거의 없었다. 그후 15년간 연구와 개발이 계속 진행되였으며 SVR4라고 하는 강력한 조작체계가 출현하였다.

벨연구소 개발자들은 Multics프로젝트에 대한 연구를 진행하여 새로운 조작체계에 영향을 주었다. 즉 Unix파일체계와 지령해석기로서 리용자프로쎄스를 리용할데 대한 문제, 파일체계대면부의 일반적인 조직화 등을 들수 있다. 새로운 프로쎄스를 생성하는 fork조작은 버클리의 GENIE로부터 도입되였다.

Unix체계가 다른 체계들과 차이나는 점은 다음과 같다.

첫째로, Unix체계는 고급언어로 씌여져있다. Unix체계에서는 체계원천코드가 아쎔블러어보다 C언어가 더 많이 씌여져있다. 실례로 결합부절환과 같은 하드웨어에 대한 호출에서는 조작체계함수들의 3%를 아쎔블리어로 리용한다.

둘째로, Unix체계는 원천코드의 형태로 배포하였다. Unix는 원천코드를 제공하여 다른 사람들이 체계를 리용하지만 내부작업을 서투르게 진행하지 못하게 하였다. 누구나 체계의 개발자이면서 리용자이므로 새롭게 자기 견해를 담을수 있다.

셋째로, Unix체계는 보다 더 값비싼 하드웨어상에서 실행하는 조작체계들에서만 볼수 있는 강력한 원시함수들을 제공한다.

초기에는 Unix체계가 PDP-11상에서 실현되였는데 이 콤퓨터는 당시로서는 값이 눅고 강력한 기계였다. 그러나 PDP-11은 작은 주소공간을 가지고있어 불편하였다. 그후 VAX-11/780과 같이 32bit 주소공간을 가진 기계가 소개되고 가상기억기와 망을 포함하여 Unix의 기능이 대폭적으로 확장되였다.

1978년 7번째 판이 출현한 후 연구그루빠는 Unix지원그룹(USG)으로 대외적인 배포물을 바꾸었다. USG는 이전에 Unix프로그람작성자의 작업환경(PWB)과 같은 체계를 내적으로 배포하고 때때로 대외적으로도 배포하였다.

7번째 판 다음의 대외적인 배포물로서는 1982년에 UNIX System III이 출현하였는데 이것은 7번째 판과 32V(VAX의 변종), 기타 Unix체계의 특징들을 포함하고있었다. 이와 함께 UNIX/RT(실시간Unix체계)특징과 PWB의 많은 특징을 포함하였다.

USG는 1983년에 System V를 안정화하여 공개하였다. 1984년 USG는 자기의 명칭을 USDL(UNIX체계개발연구소)로 바꾸고 System V Release 4를 출하하였다. 이 체계는 페지화와 쓰기복사(copy on write), 공유기억기도입 등 여러 점에서 많은 기능이 갱신되였다.

1987년 AT&T정보체계(ATTIS)를 완성하여 UNIX System V Release 8을 배포하였다. 여기서 프로쎄스사이통신(IPC)기구가 도입되였다. 사실 이것은 V8(8번째 판)로부터 도입되였다.

USL(Unix체계연구소)은 1993년 ATTIS를 완성하여 Novell회사에 판매하였다. 2년후 Novell은 Unix를 SCO(Santa Cruz Operation)에 판매하였다.

Unix의 형태는 다종다양하다.

대표적인것이 Microsoft회사와 SCO가 공동으로 개발한 XENIX이다. 이 체계는 원래 7번째 판에 기초하고있었으나 그후 System V에 기초하여 갱신되였다.

다음의 대표적인 체계는 버클리의 켈리포니아종합대학에 있는 연구집단이 개발한 버클리쏘프트웨어배포물(BSD)이 있다. 이 체계는 가상기억기와 요구페지화, 페지재배치 기능을 추가하여 1979년 BSD3을 발표하였다. 4BSD에는 인터네트망통신규약 TCP/IP를 제공하고 광대역과 여러가지 망기구에 대응하였다. 4.4BSD에서는 가상기억체계를 갖추고 68000, SPARC, MIPS, Intel PC에서 가동을 실현하였다.

2) Linux의 발전

Linux체계는 1991년 당시 필란드의 헬싱키대학 학생이였던 리누스 토발즈(Linus B. Tovalds)가 386콤퓨터에서 실행가능한 Minix와 비슷한 조작체계를 만들기 시작한데서 시작되였다고 말할수 있다. 처음에 리누스는 조작체계의 이름을 Freax라고 지으려고 하였다. MS-DOS는 보호방식등에서 386의 성질을 충분히 끌어낼수 없기때문에 리누스는 새로운 기능과 성질을 추가하기 위해 쏘프트웨어의 일부를 바꾸어 쓰기 시작하였다. 그후 그 결과를 Linux(Linus+Unix)라는 이름으로 인터네트의 새소식교환마당인 《Net News》에 무료로 공개하였다.

그는 초기판본을 0.01로서 1991년 8월에 공개하면서 공동개발을 호소하였으며 어떠한 공식성명도 하지 않았다. 공식판본은 0.02로서 1991년 10월에 발표되였다. 이 체계에서는 bash와 gcc와 같은 여러 형태의 GNU프로그람이 동작하였다.

단일CPU의 i386기계에서만 실행가능한 Linux핵심부 1.0의 공식안정판은 1994년 3월이였다. 그후 이에 호응하여 세계 각국의 쏘프트웨어개발자들이 자발적으로 참가하여 세계적범위에서 인터네트를 통한 쏘프트웨어개발체계가 구축되였다. 즉 쏘프트웨어개발자들은 체계개발에 대한 자기의 립장과 자기가 개발한 프로그람원천코드를 발표하고 애호가들은 오유를 발견하고 프로그람에 대한 평가를 진행하며 주개발자는 이러한 내용들을 집약해서 새로운 판본의 체계를 완성하여나가는 하나의 새로운 개발방법이 출현하게 되였다.

리누스는 아직까지 Linux체계를 개선하기 위한데 힘쓰고있으며 다양하게 발전하는 하드웨어를 지원하도록 하는 한편 전세계 수백명에 이르는 Linux개발자의 체계개발활동을 조종하고있다.

1.0판이 출현한 때로부터 1년후인 1995년 3월에 최초로 i386이 아닌 기반에서 실행가능한 (그러나 여전히 단일CPU에서만 동작하는)Linux 1.2가 안정화되였다. 1996년 6월에 Linux 2.0이 안정화되였다. 2.0에는 동작가능한 여러 기반이 추가되였지만 무엇보다도 다중CPU를 갖는 기계(SMP)에서 동작하는 최초의 판본이였다. 2.0의 안정판 이후 주요판본의 안정판속도는 다소 늦춰졌다. Linux 2.2판은 1999년 1월, 2.4판은 2001년 1월에 각각 안정화되였다. 하지만 각각의 판본갱신은 자주 일어나 지원되는 하드웨어의 범위와 확장성이 개선되여갔다. Linux 2.4판은 제일 처음으로 ISA plug and play와 USB, PC 카드 등의 기능을 추가하여 사용자들의 탁상판에서 사용할수 있는 판본으로 되였다. Linux 2.6은 2003년 12월 17일에 안정화되였다. 2.6은 다양한 추가기능도 기능이지만 대용량체계로부터 아주 작은 체계(PDA 등)까지 골고루 지원한다는 점에서 또 하나의 큰 개선을 가져온 판본이기도 하다.

Linux는 초기에는 개인용콤퓨터만을 대상으로 개발되였지만 현재에는 DEC, Alpha를 적재한 기계와 Sparc WS 등에서도 가동하는 기반의 폭이 대단히 넓다. 많은 개발자들이 Hewlett Packard의 Alpha나 Intel의 최신 64bit처리장치인 Itaniurn, MIPS, SPARC, Motorola MC68OxO, PowerPC, IBM z계렬 등 다양한 기본방식에서 Linux를 사용할수 있도록 노력하고있다.

Linux체계는 POSIX 1003.1과 UNIX System V, BSD 4.3의 기능을 담고있다. 사실 Windows NT처럼 Unix조작체계가 아니면서 POSIX호환인 조작체계도 여러 종류가 있다.

Linux체계는 GNU일반공개특허(GNU General Public License）의 조건에 의해 배포되며 리용자에게는 원천코드가 공개되고 수정과 복사, 배포가 자유롭다. GNU의 어원은 영문자로 볼 때 《Gnu is Not Unix》인데 여기서 Gnu는 남아프리카에서 서식하는 들소종류의 이름이다. GNU는 1983년에 무료쏘프트웨어재단(FSF)의 창시자인 리챠드 스텔만(Richard Stallman)에 의해 시작된 큰 프로젝트이며 그때 당시 프로젝트의 이름은 《RMS》라고 하였다. 그는 강력한 본문편집기인 emacs를 제작하였으며 MIT의 인공지능연구소에서 지도적인 지위를 맡고있다. 그는 무료로 배포되는 조작체계의 원천코드를 만들것을 목적으로 큰 규모의 프로젝트를 시작하기로 결심하고 체계에 필요한 콤파일러로부터 본문편집기까지 모든 체계도구들을 만드는 방향으로 나가고있다.

Linux는 GNU가 아니라 조작체계이며 GNU의 조작체계는 Linux가 아니라 《Hurd》라는 이름으로 개발되고있다. 기술적으로 따져보면 Linux는 Unix핵심부이지만 완전한 Unix조작체계는 아니다. Linux에는 파일체계편의프로그람이나 Windows체계, 도형탁상판, 체계관리명령어, 본문편집기, 콤파일러 같은 Unix응용프로그람이 없

기때문이다. 하지만 이 프로그람들은 GNU일반공개특허에 따라 자유롭게 사용할수 있기때문에 Linux가 지원하는 파일체계에 설치할수 있다.

Linux핵심부는 크게 개발단계와 안정화단계로 나누어 개발되고있으며 이 두 단계의 개발이 동시에 진행된다.

개발단계(개정단계라고도 한다.)에서는 핵심부의 믿음도보다는 새로운 기능의 추가를 중시한다. 여기서는 새로운 착상을 시험해보기 위해 핵심부를 최적화하기도 하고 기능을 추가하기도 한다. 핵심부판본의 두번째 수자가 홀수이면 이 핵심부는 개정판핵심부로 된다. 즉 2.1, 2.3, 2.5등으로 증가한다. 실례로 핵심부판본이 2.5.62라고 할 때 2는 판본번호이고 5는 홀수로서 개정판을 나타내며 62는 이 개정판이 62번째로 공개되였다는것을 나타낸다. 이 단계에서는 핵심부에 대해서 가장 많은 작업이 진행된다.

안정화단계에서는 될수록 핵심부를 안전하게 하는것을 목적으로 한다. 여기서는 최소한의 수정과 조정이 진행된다. 핵심부판본은 두번째 수자가 짝수로 증가한다. 즉 2.2, 2.4, 2.6 등으로 증가한다. 실례로 2.6.9라고 할 때 안정판핵심부 2.6이 9번째로 공개되였다는것을 의미한다.

2. Linux핵심부의 특징

Linux는 가상기억기, 가상파일체계, 《가벼운》프로쎄스, 믿음성신호기, SVR4의 프로쎄스사이의 통신, 대칭형다중처리기에 대한 지원 및 체계보안 등의 우월한 기능들을 실현하고있다.

Linux핵심부는 여러가지 론리적인 구성요소로 이루어진 복잡하고 비대하며 독립적인 프로그람이다. 전통적으로 대부분의 Unix핵심부는 단일묶음방식（monolithic）핵심부이다.

Linux는 모듈을 단위로 동적으로 적재하거나 해제할수 있다. 대체로 Linux에서는 파일체계와 구동프로그람들을 모듈로 관리하고있다. 이 기능은 핵심부 2.2판부터 제공하기 시작하였는데 기억기관리에서 효률적이다. 특히 Unix체계에 비해 요구에 따라 자동적으로 적재 및 해제할수 있는 모듈지원기능이 갖추어져있다. 주요상용Unix변종중에서는 유일하게 SVR4.2와 Solaris핵심부에 이와 류사한 기능이 있다.

Linux는 독립적으로 순서짜기할수 있는 실행결합부로서 핵심부스레드기능을 지원하고있다. 이와 함께 다중스레드식응용프로그람을 지원함으로써 응용프로그람구조상 큰 부분들을 공유할수 있는 응용프로그람을 개발할수 있게 된다. Solaris2.x나 SVR4.2/MP 같은 몇몇 최신 Unix핵심부도 핵심부스레드(kernel thread)집합으로 이루어져있다. 핵심부스레드는 사용자프로그람과 련결되기도 하고 몇가지 핵심부함수만을 실행하기도 한다. 핵심부스레드사이의 결합부절환(context switch)은 대역주소공간에서 이루어지기때문에 보통 일반 프로쎄스사이의 결합부절환에 비해 비용이 훨씬 적게 든다. Linux에서는 핵심부의 기능중 몇가지를 주기적으로 실행하는 매우 제한된 목적에

만 핵심부스레드를 사용한다. Linux핵심부스레드는 사용자프로그람을 실행할수 없기때문에 기본적인 실행결합부개념을 나타낸다고 할수 없다.

대부분의 현대조작체계는 다중스레드(multithread)응용프로그람 즉 응용프로그람에 있는 자료구조의 상당한 부분을 공유하면서 상대적으로 독립적인 여러 실행흐름을 통해 작업하도록 설계된 프로그람을 지원한다.

다중스레드 사용자응용프로그람은 많은 《가벼운 프로세스 (LWP:lightweight process)》로 이루어진다. 가벼운 프로세스는 대역주소공간, 대역물리기억페지, 대역열린파일등에서 동작하는 프로세스이다.

Linux는 SVR4나 Solaris 같은 체계에서 사용하는것과는 다른 독자적인 가벼운 프로세스를 실현한다. 상용Unix변종의 가벼운 프로세스는 모두 핵심부스레드에 기초하고 있지만 Linux는 가벼운 프로세스를 기본적인 실행결합부로 여기고 비표준인 clone()체계호출(system call)을 통해 리용한다.

Linux핵심부는 다중처리장치도 지원하고있다. 즉 같은 처리장치가 여러개 존재하는 대칭형다중처리기(SMP)도 지원하고있다. SMP체계에서는 여러 처리장치를 사용할수 있고 각 처리장치가 어떤 작업이든 할수 있으며 모든 처리장치가 서로 동등하다.

Linux에는 각이한 특징을 가진 표준 파일체계들이 존재한다. 특별한 요구가 없다면 이중에서 가장 일반적인 Ext2파일체계를 사용할수 있다. 체계가 꺼지고 다시 켜질때마다 파일체계검사가 동작하는것이 싫다면 Ext3파일체계로 바꿀수도 있다. 작은 파일들을 많이 다루어야 한다면 ReiserFS파일체계가 가장 효과적이다. Ext3이나 ReiserFS외에 Linux원천코드에는 들어있지 않지만 몇가지 기록형(journaling)파일체계를 사용할수도 있다. 이러한 파일체계로서는 IBM AIX의 기록형파일체계(JFS, Journaling F1le System)와 SGI lrix의 XFS파일체계가 있다. 강력한 객체지향가상파일체계기술(Solaris와 SVR4에서 제안)에 의해서 Linux에서는 다른 조작체계의 파일체계를 Linux용으로 전환하는 작업은 상대적으로 아주 쉽다.

Linux는 무료로 배포되므로 하드웨어와는 달리 비용을 전혀 들이지 않고도 설치를 할수 있으며 모든 구성부분들을 마음대로 변경할수 있다.

Linux핵심부는 요구되는 하드웨어준위가 다른 체계들에 비해 아주 낮다. 즉 주기억기가 4MB이고 CPU가 80386인 장치조건에서도 망봉사기를 구축할수 있다.

Linux핵심부는 크기를 작게 할수 있으며 압축도 가능하다. 실례로 모든 기본체계프로그람기능을 다 포함한 핵심부와 완전한 뿌리파일체계를 1.4MB의 유연성디스크 1장에 구성할수 있다.

Linux는 가동기반의 폭이 대단히 넓다. Linux는 개인용콤퓨터들에 가장 많이 쓰이고있는 Intel계렬의 x86(386, 486, Pentium계렬, Itanium 등)의 처리장치뿐아니라 AMD나 Cyrix와 같은 Intel호환처리장치들도 지원하고있다. 이외에 Alpha, SUN Sparc, PowerPC, Crusoe, ARM, MC680x0 등 각이한 기반의 처리장치들을 지원

하고있다.

Linux는 일반 다른 조작체계들과 호환을 가진다. 실례로 MS-DOS, MS-Windows, SVR4, OS/2, MacOS, Solaris, SunOS 등 여러 조작체계들과 파일체계 준위와 코드준위에서 일정한 호환을 가지고있다.

Linux핵심부는 그 원천코드의 질이 대단히 높으며 개발속도도 빠르다. 핵심부는 그 원천코드가 완전히 공개되여 세계의 많은 프로그람작성자들과 애호가들에 의해 인터네트를 통하여 개발되고 수정되기때문에 그 갱신속도가 빠르고 원천코드의 질도 높은 수준에서 보장되고있다. 매일매시각 새로운 프로그람들과 기능들이 발표되고 기존의 프로그람들이 매우 빨리 수정되고있다. 만일 핵심부기능에서 치명적인 오유나 보안구멍이 발견되였다면 인차 그 문제를 해결하는 패치(수정프로그람)가 원천코드로 발표되며 이것이 다음번 판본에 반영되게 된다.

Linux핵심부2.6은 기업정보체계의 기능을 강화하고 체계의 확장성을 높이는데 기본을 두고 개발되였다. 이와 함께 과학기술계산을 위한 클라스터체계와 매몰기구, 노트형PC 등 모바일환경을 지향하여 많은 기능을 제공하고있다.

체계확장성의 향상

다중처리소자구성에서의 처리의 병렬도가 더욱 높아지고 다중처리소자의 배타조종기구가 성능이 향상되였다.

핵심부2.6에서는 새롭게 《RCU :Read Copy Update》라고 하는 배타기구와 《Sequence counter》라고 하는 배타방식을 도입하였다. 이것은 종래의것보다 체계의 확장성을 높이는 기술로서 기대되고있다.

한편 핵심부2.6에서는 그 전까지 완전히 병렬처리할수 있게 바꾸어쓰지 못하였던 코드를 모두 병렬처리를 할수 있는 새로운 코드로 써넣었다. 그 대부분이 장치구동기의 코드로 되고있다.

핵심부2.6에서는 많은 프로쎄스가 동시에 실행할 때 처리시간이 최소로 되게 프로쎄스순서짜기프로그람이 새로운것으로 바뀌여졌다. 즉 프로쎄스의 실행우선도마다에 대기렬을 두고 그 대기렬중에서 가장 높은 우선도를 가진 대기렬의 선두로부터 차례로 실행가능한 프로쎄스를 선택한다. 이렇게 최소한의 검색량으로 차례로 실행하는 프로쎄스를 선택할수 있으며 실행가능프로쎄스가 여러개 동시에 존재하여도 성능이 떨어지지 않는다. 그리고 될수록 다중처리소자구성에서 각 CPU부하상태를 검사하고 부하가 클 때에만 프로쎄스를 다른 CPU에로 이동시키게 하므로 CPU사이에서 프로쎄스의 순서짜기에 걸리는 시간을 감소시키고있다.

핵심부 2.6에서는 NUMA(Non-Uniform Memory Access)에 대응시키고있다. 사실 종래의 대칭형다중처리기(SMP)에서는 CPU수가 증가할수록 성능이 떨어진다. 따라서 처리소자수를 증가시키려는 경우에는 SMP가 아니라 SMP기계를 고속모선으로 결합하는 구성을 가진 비대칭형다중처리소자체계인 NUMA을 받아들이고있다.

이렇게 되면 고성능인 대규모콤퓨터를 비교적 낮은 가격으로 실현할수 있다.

파일체계처리의 개선

핵심부2.6의 최대의 특징이라고 할수 있는 기능은 파일체계와 블로크I/O의 처리효률이 상당히 개선된것이다. 이것은 대규모체계에서 성능을 떨구지 않게 해주는 기능으로 된다.

블로크에로의 입출력처리는 그의 순번조종과 입출력요구의 무리체계(clustering)에 의해 높은 성능을 발휘하게끔 되여있다. 종래의 판본에서는 입출력처리가 완충기(완충기캐쉬)단위로 진행되였지만 핵심부 2.6에서는 페지(페지캐쉬)단위로 진행되게 되였으며 캐쉬는 모두 페지캐쉬로 통합되고 완충기캐쉬는 없어졌다. 따라서 완충기캐쉬에 대해서 진행되던 자료분할과 통합작업이 생략되고 보다 처리를 고속으로 진행할수 있게 되였다. 그리고 디스크에로의 지연쓰기는 파일단위로 진행할수 있게 되였다.

파일입출력처리의 병렬도를 확장하였다. 파일체계와 블로크입출력계층의 배타조종을 전면적으로 진행할수 있게 되였다.

핵심부 2.6에서는 대용량의 기억기를 효률적으로 관리할수 있는 기능이 추가되였다. 여기서는 상위기억구역이라고 해도 DMA로 리용할수 있는 령역이면 기억기와 하드웨어장치사이에서 직접 자료를 전송할수 있게 되여있다.

핵심부 2.6에서는 I/O순서짜기를 예측과 배합하여 진행하고있다. 새로운 알고리듬은 《1개의 프로쎄스는 계속하여 린접한 블로크에 대하여 같은 종류의 입출력요구를 내보내려는 경향이 있다.》라는 가정에 기초하여 순서짜기를 진행하고있다. 즉 련속인 입출력요구가 있는경우 그 시점에서 이에 계속하여 입출력요구가 없어도 약간 기다리면 련속인 입출력요구가 발생할지도 모른다고 생각할수 있다. 이 입출력처리에는 예측순서짜기프로그람외에 차단선(deadline)순서짜기알고리듬이 실현되여있다.

아주 큰 자료에 대해서도 취급할수 있게 성능이 많이 개선되였다. 즉 크기가 2GB를 넘는 파일과 2TB를 넘는 파일체계를 취급할수 있다.

망파일체계인 NFS v3의 성능을 대폭 확장하고 NFS v4도 지원하고있다.

자료기지지향의 기능

핵심부2.6에서는 자료기지의 리용을 전제로 하여 기능을 대폭 확장하고있다. 즉 독자적인 캐쉬기능을 가지고 자체로 자료를 관리하고있는 자료기지쏘프트웨어의 호출특성에 맞게 기능을 확장하였다.

벡토르읽기/쓰기가 여러 완충기에서 진행되며 처리를 묶어서 1번에 진행하므로 입출력처리의 회수가 감소되였다.

핵심부가 관리하는 페지고속완충기를 거치지 않고 응용프로그람이 직접 파일에 호출하는 직접입출력기구는 무리체계기구와 벡토르읽기/쓰기에 의해 효률적으로 동작하게 된다.

비동기읽기/쓰기는 종래에는 서고준위에서 보장하였지만 핵심부 2.6에서는 핵심부준위에서 비동기I/O기능을 제공한다.

파부하시의 견딤성의 향상

종래의 핵심부판본에서는 체계가 파부하로 되였을 때 성능이 극단으로 떨어졌지만 핵심부2.6에서는 그러한 파부하상태에서도 일정한 성능을 유지할수 있게 되여있다.

페지캐쉬상의 페지와 프로쎄스공간용으로 리용하고있는 페지의 참조빈도수에 대한 비교를 간단하게 만들므로써 파부하시에도 페지해방을 효과적으로 진행하며 일정한 성능을 유지할수 있게 한다. 또한 교환크기이상으로 프로쎄스공간이 확대되는것을 금지하며 응용프로그람이 폭주하여 교착상태에 빠질 위험성도 감소시키고있다.

파일입출력처리의 부하가 높은 상태에서는 입출력요구가 가장 오랜 대기시간을 정의하여 그 시간을 초과한 입출력요구를 우선적으로 처리함으로써 종전에 비해 입출력응답성을 높이고있다.

망부하가 클 때 망구동기를 새치기구동으로부터 문의구동으로 바꾸어줌으로써 필요이상의 새치기가 발생하지 않도록 억제함으로써 처리시간을 가장 작게 한다.

가용성의 향상

체계에서 될수록 봉사를 제공할수 없는 시간을 짧게 하고 봉사시간을 높이기 위한 기능이 강화되였다.

핵심부 2.6은 용도에 따라 네가지 종류의 기록형파일체계로부터 선택하여 리용할수 있다. 여기서는 Linux용으로 개발된 Ext3와 ReiserFS와 함께 상용UNIX로부터 이식된 JFS와 XFS도 리용할수 있다.

동적파일체계확장기능을 가지고있다. 정기적으로 체계의 운영을 해나갈 때에는 하드디스크 등 각종 자원을 확장하여야 하는데 핵심부 2.6에서는 이 경우 파일체계를 리용하고있는 상태에서 핵심부가 인식하는 Volume정보를 갱신하고 파일체계의 크기도 갱신할수 있게 되여있다.

망기능의 강화

핵심부 2.6에서는 IPv6의 기능이 종전의 판본에 비하여 대폭 강화되였다. USAGI프로젝트(Linux용의 완전한 IPv6의 실현을 목적으로 한 프로젝트)의 성과들이 도입되고 완성도도 높아지고있다.

핵심부 2.6에서는 UDP와 TCP의 중간층통신규약인 SCTP을 제공하고있다.

핵심부 2.6에서는 통신을 암호화하는 기술의 하나인 IPSec기능을 제공한다. 이 기능은 IPv4와 IPv6에서 다 같이 리용할수 있다.

핵심부 2.6에서는 IP의 적재균형기능인 IPVS가 도입되였다. 따라서 부하분산형의 무리체계를 짜넣는것이 가능하다.

보안기능의 강화

Linux핵심부 2.6에서는 Linux보안모듈이 추가되여 보안기능이 대폭 강화되였다. 전통적인 UNIX체계에서의 root권한이 모든 파일에 대하여 접근권한을 가지고 있었지만 Linux에서는 프로쎄스마다에 한정된 권한을 제공하는 기능으로서 체계확장성을 위한 구조가 도입되고있다. 핵심부2.6에서는 이 구조를 다시 확장하고 용도에 따라서 각이한 접근조종방책을 핵심부에 짜넣을수 있게 하였다. 구체적으로 핵심부내의 각 오브젝트를 조작하는 개개의 개소마다 접근의 가부를 판정하는 함수를 짜넣고있다.

핵심부 2.6에서는 소유자의 리용자ID와 그룹ID외에 POSIX ACL형태의 접근조종도 가능하게 되여있으므로 1개의 파일에 접근할수 있는 여러사용자와 여러그룹을 정의하고 다시 개개의 리용자와 그룹마다에 접근권한을 설정할수 있게 된다.

핵심부 2.6에서는 핵심부내에서 각종 암호화처리함수를 갖추고있다. 현재 HMAC, MD4, MD5, SHA-1, DES, Triple DES EDE, Blowfish, AES등이 제공되여있다.

핵심부 2.6에서는 이름공간을 프로쎄스마다 준비하여 그 프로쎄스용으로 파일나무를 파일체계에 태우기할수 있게 되여있다. 실례로 FTP봉사기등으로 필요한 등록부밖에 보이지 않게 하며 특정한 프로쎄스로 제한된 이름공간만을 제공하게 하는 등의 목적에 리용할수 있다.

전원관리기능의 강화

종전의 핵심부판본에서는 ACPI(Advanced Configuration and Power Interface)의 일부 기능밖에 리용할수 없었지만 핵심부2.6에서는 대체로 모든 기능을 리용할수 있게 되여있다. 또한 Windows와 같이 쏘프트웨어 suspend라고 하는 기능이 제공되고 또한 CPU의 동작주파수를 동적으로 변경할수 있게 되여있다.

매몰기구지향의 기능

핵심부2.6에는 종전의 uCLinux라고 하는 명칭으로 별도로 개발되고있던 Linux의 코드가 통합되여 조립용도의 생전력/생공간형의 CPU에 정식으로 대응할수 있게 되여있다. 이와 함께 MMU기능이 없는 조립용도지향의 CPU를 동작시킬수 있게 되여있다.

장치구동기의 확장

핵심부2.6에서는 수많은 장치구동기를 제공할수 있게 되여있다.

새롭게 USB2.0규격에 대응하였다. 종래의 저속(1.5Mbit/s), 전속(12Mbit/s)에 고속(480Mbit/s)의 전송속도를 보장하는것이 추가되였다.

종래의 체계의 OSS(Open Sound System)음성구동기대신에 ALSA (Advanced Linux Sound Architecture)구동기가 도입되였다. 이 구동기는 눅은 가격의 음성카드로부터 다중통로를 가진 음성대면부까지 폭넓게 제공하고있다. 또한 OSS호환대면부도 갖추고있으므로 지금까지 있던 음성관련응용프로그람도 그대로 리

용할수 있다.

고속으로 묘화를 진행할수 있는 AGP3.0규격에 대응한 구동기를 제공하고있다. 따라서 현재 시장에 등장하고있는 이 규격의 비데오카드를 리용할수 있다.

핵심부2.6에서는 새롭게 광범히 리용되기 시작한 각종 기가비트 이써네트카드에 대응하고있다.

제 2 절. 조작체계의 기초

모든 콤퓨터체계에는 《조작체계》라고 하는 기본적인 프로그람집합이 들어있고 이 프로그람집합에서 가장 중요한 역할을 담당하는 프로그람을 《핵심부(kernel)》라고 한다. 핵심부는 체계가 초기기동할 때 BIOS에 의해 주기억기에 적재되여 사용자응용프로그람과 콤퓨터장치사이의 결합을 가장 아래준위에서 실현하는 핵심프로그람이다. 바로 체계의 본질적인 형태와 성능은 이 핵심부에 의해 좌우된다. 핵심부는 하드웨어와 웃준위쏘프트웨어에 대한 조종과 그에 대한 요구처리를 진행함으로써 전체적인 콤퓨터체계가 사용자의 요구에 충실히 복무하도록 한다. 그리하여 때때로 핵심부자체를 가리켜 조작체계라고도 한다.

Linux와 Windows 등 현대적인 조작체계는 하드웨어와 조작체계의 구조를 몰라도 간단히 응용프로그람을 작성할수 있는 환경을 제공하고있다. GUI가 원만하여 리용자에 있어서 리용하기 쉬운 환경으로 되여있다. 자동차에 비교하면 초기 자동차가 등장한 때에는 그 구조를 잘 알고있는 사람밖에 리용할수 없었지만 현재는 자동차의 내부구조를 깊이 알고있지 않아도 간단히 운전할수 있는 기구가 갖추어져있다. 그러나 자동차의 성능을 최대한 리용하려고 하는 경우 내부구조에 대해 완전히 알아야 한다. 마찬가지로 콤퓨터의 성능을 최대로 발휘하기 위해서는 리용자에 대하여서도 숨겨져있는 부분까지도 리해할 필요가 제기된다. 그래야 체계의 약점으로 되는 부분을 발견할수 있고 보다 효률적인 응용프로그람을 개발할수 있게 된다.

조작체계의 기본기능은 하드웨어기반을 이루고있는 프로그람조종가능한 모든 저준위장치구성요소들을 봉사하는 하드웨어구성부분들과 호상작용하면서 콤퓨터체계에서 실행되고있는 응용프로그람들에 대한 실행환경을 보장하는것이다.

1. 핵심부구조

앞에서도 이야기 한것처럼 대부분의 Unix핵심부는 단일묶음방식이다. 다시말하여 개개의 핵심부계층이 핵심부프로그람 하나에 통합되여있으며 현행프로세스를 대신하여 핵심부방식에서 동작한다. 이와 반면에 《마이크로핵심부(microkernel)》조작체계는 일반적으로 몇가지의 동기화원시함수와 간단한 순서짜기프로그람, 프로세스사이의 통신

기구을 포함하여 매우 적은 기능만을 핵심부에 요구한다. 마이크로핵심부에서는 일부 체계프로쎄스를 실행하여 기억기할당이나 장치구동프로그람, 체계호출운전기 등 다른 조작체계계층에 들어있는 기능을 실현한다.

조작체계분야에서 학술적인 연구가 마이크로핵심부방향으로 치우쳐 있지만 마이크로핵심부조작체계의 여러계층사이에 통보문을 전달하는데 지내 품이 많이 들므로 일반적으로 단일묶음방식핵심부보다 속도가 뜨다. 그러나 마이크로핵심부조작체계에서는 단일묶음방식조작체계에 비해 리론적으로 몇가지 우점을 가지고있다. 마이크로핵심부의 각 조작체계계층은 잘 정의된 쏘프트웨어대면부를 통해서만 다른 계층과 호상작용할수 있는 비교적 독립인 프로그람이기때문에 체계프로그람작성자들이 모듈화된 접근을 하도록 한다. 더우기 현재의 마이크로핵심부조작체계는 하드웨어에 의존하는 모든 구성요소를 마이크로핵심부코드안으로 집어넣었기때문에 다른 기본방식에 이식하기도 아주 쉽다. 그리고 마이크로핵심부조작체계에서는 불필요한 기능을 수행하는 프로쎄스를 바꾸어내보내거나(swap out) 없앨수도 있으므로 단일묶음방식조작체계보다 주기억기를 효과적으로 리용할수도 있다.

Linux핵심부는 성능을 떨어뜨리지 않고 마이크로핵심부의 여러가지 우점을 살리기 위해 《모듈(module)》을 제공한다. 모듈은 실행중인 핵심부에 련결될수 있는 (련결해제도 가능) 오브젝트파일이다. 이 오브젝트코드는 대체로 파일체계와 장치구동프로그람, 핵심부의 웃준위에 있는 다른 기능을 실현하는 함수의 집합으로 되여있다. 모듈은 마이크로핵심부조작체계의 외부계층과는 달리 개개의 프로쎄스로 동작하지 않는다. 대신에 모듈은 정적으로 련결된 다른 핵심부함수와 마찬가지로 현행프로쎄스를 대신하여 핵심부방식에서 동작한다.

모듈을 리용하면 여러가지 우점이 있다.

우선 모듈화된 접근을 할수 있다. 모든 모듈은 실행중에 련결이나 련결해제를 할수 있기때문에 체계프로그람작성자는 모듈이 다루는 자료구조에 접근할수 있도록 잘 정의된 쏘프트웨어대면부를 제공하여야 한다. 그래야 새로운 기능을 실현하는 모듈을 쉽게 개발해나갈수 있다.

다음으로 기반에 의존하지 않게 할수 있다. 모듈은 특정한 하드웨어에 국한된 기능을 사용할수도 있지만 보통은 고정된 하드웨어기반에 의존하지 않는다. 실례로 SCSI표준에 맞게 작성된 디스크구동모듈은 IBM콤퓨터만이 아니라 Alpha기계에서도 잘 동작한다.

또한 모듈을 사용하면 주기억기를 적게 사용한다. 모듈은 그 기능이 필요한 때 실행중인 핵심부에 련결할수 있으며 더 이상 필요 없으면 련결해제할수도 있다. 핵심부는 자동적으로 련결과 해제를 진행하면서 기억기의 효률을 높인다.

모듈을 사용하면 성능을 유지할수 있다. 일단 한번 련결되면 모듈의 오브젝트코드는 정적으로 련결된 핵심부의 오브젝트코드와 동등해진다. 따라서 모듈에 있는 함수를 호출

할 때에는 별도의 통보문전달이 필요없다.

2. 파일체계개요

파일체계는 자료관리의 제일 핵심적인 부분으로서 핵심부기능에서 중요한 자리를 차지한다. 여기서는 Linux체계에서 파일관리에 대해서 개괄적으로 보기로 한다.

1) 파일

일반적으로 파일이라고 할 때 여러 바이트로 이루어진 정보를 담는 그릇이라고 생각할수 있다. 핵심부는 파일의 내용이 무엇인지는 신경쓰지 않는다. 많은 프로그람서고는 마당(field)으로 이루어진 레코드(record)나 열쇠를 기초로 하여 레코드지정과 같은 높은 수준의 추상화를 실현한다. 그러나 이러한 서고에 있는 프로그람은 핵심부가 제공하는 체계호출을 사용하여야 한다. 사용자의 관점에서 보면 파일은 아래에서 보는것처럼 나무형태의 이름공간으로 조직되여있다.

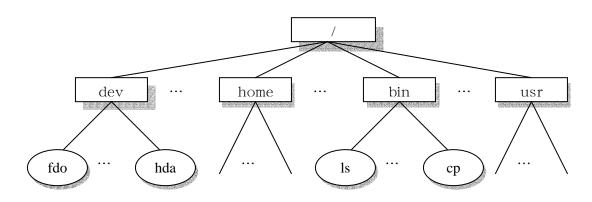



그림 1-1. 리눅스파일체계의 나무구조

나무에서 잎(leaf)을 제외한 모든 마디(node)는 등록부이름을 의미한다. 각 등록부마디는 바로 아래에 있는 등록부와 파일에 관한 정보를 포함하고있다. 파일이나 등록부명은 /과 빈(null)문자인 \0을 제외한 일련의 아스키(ASCII)문자이다. 대체로 파일체계들에서는 파일명의 길이에 제한을 가지는데 보통은 255문자를 넘지 않는다. 나무의 뿌리에 해당하는 등록부를 《뿌리등록부(root directory)》라고 한다. 전통적으로 뿌리등록부명은 빗선(/)이다. 같은 등록부안에 있는 이름들은 서로 달라야 하지만 서로 다른 등록부에서는 같은 이름을 사용할수 있다.

Linux에서는 각 프로쎄스가 《 현재의 작업등록부(current working directory)》와 련관된다. 이것은 프로쎄스의 실행결합부의 일부로서 프로쎄스가 현재 사용하는 등록부를 가리킨다. 프로쎄스는 특정의 파일을 지정할 때 《경로명(빗선과 등록부명이 엇바뀌면서 파일명을 지정한것)》을 사용한다. 경로명이 빗선으로 시작하면 그

경로명은 시작지점이 뿌리등록부이기때문에 《절대(absolute)경로》라고 한다. 그렇지 않고 등록부명이나 파일명으로 시작하는 경우 그 경로명은 시작지점이 프로쎄스의 현재 등록부이기때문에 《상대(relative)경로》라고 한다.

파일명을 지정할 때 《.》과 《..》을 사용하기도 하는데 각각 현재의 작업등록부와 그 부모등록부를 의미한다. 현재의 작업등록부가 뿌리등록부인 경우 《.》과 《..》은 꼭같다.

2) 하드련결과 쏘프트련결

등록부에 있는 파일명을 《파일하드련결(file hard link)》, 더 간단히 줄여서 그냥 련결이라고 한다. 똑같은 파일에 대해 같은 등록부나 다른 등록부에서 여러개의 련결을 가짐으로써 한 파일이 여러 파일명을 가질수 있다.

아래의 명령은 경로명 f1이 나타내는 파일에 대해 f2이라고 하는 경로명으로 새로운 하드련결을 만든다.

```
$ ln f1 f2
```

하드련결에는 두가지 제한이 있다.

우선 등록부에 대한 하드련결을 만들수 없는데 이것은 나무구조인 등록부계층구조를 순환고리로 만들어 파일명으로 파일을 찾지 못하게 할수도 있다.

다음으로 같은 파일체계에 들어있는 파일사이에서만 하드련결을 만들수 있다. 이것은 최근의 Linux체계는 서로 다른 디스크나 다른 구획에 위치하는 여러 파일체계를 소유할수 있고 사용자는 그것들이 물리적으로 어떻게 구분되여있는지 모를수 있기때문에 심중한 문제이다.

쏘프트련결(soft link)[기호련결(symbolic link)이라고도 한다.]은 이러한 제한을 극복하기 위해 도입된것이다. 기호련결은 다른 파일의 경로명을 포함하는 작은 파일이다. 경로명은 아무 파일체계에 들어있는 어떠한 파일이라도 가리킬수 있다. 지어 실제로 존재하지 않는 파일도 가리킬수 있다.

아래의 명령은 경로명 f2가 경로명 f1을 가리키도록 소프트련결을 만든다.

```
$ ln  s f1 f2
```

이 명령을 실행하면 파일체계는 f2에 해당하는 등록부에 f2로 지정한 이름으로 기호련결형태의 입구점을 새로 만들고 f1의 경로명을 파일내용으로 포함하도록 한다. 이렇게 하여 f2을 참조하면 자동으로 f1을 참조하도록 한다.

3) 파일형태

파일에는 일반파일, 등록부, 기호련결, 블로크장치에 해당한 장치파일, 문자형장치에 해당한 장치파일, 관(pipe)과 이름있는 관(named pipe, FIFO라고도 한다.), 소케트라는 7가지 형태가 있다. 구체적인것은 앞으로 구체적으로 고찰한다.

4) 파일서술자와 i마디

Linux는 파일의 내용과 파일과 관련된 정보를 명확하게 구별한다. 장치파일과 특

수파일을 제외하고 각 파일은 일련의 문자로 이루어진다. 파일에는 파일의 길이나 파일의 끝(EOF: End Of File)을 나타내는 구분자와 같은 조종정보가 들어 있지 않다.

파일체계가 파일을 다루는데 필요한 모든 정보는 《i마디(inode)》라는 자료구조에 들어있다. 각 파일은 자기만의 i마디를 가지며 파일체계는 파일을 구별할 때 이 i마디를 사용한다. 파일체계와 파일체계를 다루는 핵심부함수는 Unix체계마다 크게 차이가 있을 수 있지만 적어도 POSIX표준에서 규정하는 다음과 같은 속성을 항상 제공하여야 한다.

- 파일형태
- 파일과 련결된 하드련결의 개수
- 바이트단위의 파일길이
- 장치ID(파일을 포함하고있는 장치의 식별자)
- 파일체계에 들어있는 파일을 구별할수 있는 i마디번호
- 파일을 소유하고있는 사용자의 ID
- 파일의 그룹ID
- i마디상태가 언제 바뀌였는지, 언제 마지막으로 접근했는지, 언제 마지막으로 수정했는지를 나타내는 여러가지 시간기록
- 접근권한과 파일방식

5) 접근권한과 파일방식

파일을 사용하는 사용자는 다음 세개의 부류로 구분할수 있다.

- 파일을 소유하고있는 사용자
- 소유자를 제외한 파일과 같은 그룹에 속한 사용자
- 나머지사용자(그 밖의 사용자)

이 세가지 부류에 대하여 각각 읽기(read), 쓰기(write), 실행(execute)이라고 하는 세가지의 접근권한형태가 있다. 이 밖에 SUID(Set User ID: 사용자ID설정), SGID(Set Group ID: 그룹ID설정), sticky라는 세가지의 추가적인 기발이 파일방식을 정의한다. 실행파일이 이 기발을 가지면 각각 다음과 같은 의미를 나타낸다.

SUID- 어떤 파일을 실행하는 프로쎄스는 보통 프로쎄스소유자의 UID(User ID: 사용자ID)을 가진다. 그러나 SUID기발이 설정된 실행파일을 실행하면 프로쎄스는 파일소유자의 UID를 가진다.

SGID- 어떤 파일을 실행하는 프로쎄스는 프로쎄스그룹의 GID(그룹ID, Group ID)를 가진다. 그러나 SGID기발이 설정된 실행파일을 실행하면 프로쎄스는 파일그룹의 ID를 가진다.

Sticky- sticky기발이 설정된 실행파일은 실행을 끝낸 후에도 프로그람을 기억기에 유지하도록 핵심부에 요청한다.

프로쎄스가 파일을 생성하면 파일의 소유자ID는 해당 프로쎄스의 UID가 된다. 파일의 소유그룹ID는 부모등록부의 sgid기발값에 따라 해당 파일을 생성한 프로쎄스의

GID나 부모등록부의 GID중 하나가 된다.

6) 파일관련체계호출

사용자가 일반 파일이나 등록부의 내용에 접근하면 실지로는 하드웨어의 블로크장치에 보관되여있는 자료에 접근하게 된다. 이런 의미에서 파일체계는 하드디스크구획의 물리적인 구조를 사용자관점에서 보는것이라고 할수 있다. 사용자방식에서 동작하는 프로쎄스는 저수준의 하드웨어구성요소와 직접 호상작용할수 없기때문에 실제의 파일을 다루는 작업은 핵심부방식에서 수행되여야 한다. 이때문에 Linux조작체계에서는 파일조작과 관련된 여러가지 체계호출을 정의하고있다.

모든 Unix핵심부는 전반적인 체계성능을 향상시키기 위해 하드웨어블로크장치를 효률적으로 다루는데 초점을 두고있다. 이 조작들은 체계호출과 밀접한 관련를 가진다.

파일열기

프로쎄스는 《열린》파일에만 접근할수 있다. 프로쎄스는 다음과 같이 체계호출을 통하여 파일을 연다.

fd = open(path, flag, mode);

path는 열려는 파일의 경로명(상대적 혹은 절대적)을 나타낸다.

flag는 파일을 어떻게 열것인가를 나타낸다. 실례로 읽기, 쓰기, 읽기/쓰기, 추가가 있다. 여기서 파일이 실제로 존재하지 않는 경우 파일을 새로 생성할것인지도 지정할수 있다.

mode는 새로 생성할 파일의 접근권한을 지정한다.

이 체계호출은 열린파일객체를 생성하고 파일서술자(file descripter)라고 하는 식별자를 돌려준다. 열린파일객체는 다음의 정보를 포함한다.

- 파일자료를 복사할 핵심부완충기령역에 대한 지적자, 파일에 대한 다음 연산을 수행할 파일에서의 현재 위치를 나타내는 offset마당(파일지적자이라고도 한다) 같은 파일을 다루는 몇가지 자료구조
- 프로쎄스가 호출할수 있는 핵심부함수를 가리키는 몇몇 지적자. flag파라메터의 값에 따라 사용이 허가된 함수의 목록이 달라진다.

파일서술자는 프로쎄스와 열린 파일사이의 호상작용을 나타내는 반면에 열린파일객체는 이 호상작용과 관련된 자료를 포함한다. 동일한 프로쎄스내에서 여러 파일서술자를 통해 똑 같은 열린파일객체를 식별할수도 있다.

한편 여러 프로쎄스가 동시에 같은 파일을 열수도 있다. 이 경우에 파일체계는 각각 별도의 파일서술자와 열린파일객체를 할당한다. 이런 상태에서 여러 프로쎄스가 같은 파일에 입출력을 요구하는 경우 Unix파일체계는 그 사이에 어떠한 종류의 동기화도 제공하지 않는다. 그러나 프로쎄스 스스로 전체 파일이나 파일의 일부분을 동기화할수 있도록 flock()을 비롯한 여러 체계호출을 제공한다.

새 파일을 만들려고 한다면 create()체계호출을 사용할수도 있다. 핵심부는

create()체계호출을 open()과 똑같이 관리한다.

열린파일에 접근하기

일반 Unix파일은 순차적으로 접근할수도 있고 임의의 위치를 지정하여 접근할수도 있다. 반면에 장치파일이나 《이름있는 관(named pipe)》은 대체로 순차적으로 접근한다. 어떤식으로 접근하든 핵심부는 파일지적자를 열린파일객체에 보관한다.

read()와 write()체계호출은 항상 현재 파일지적자의 위치를 참조한다. 즉 기정적으로 순차적인 접근으로 생각한다. 이 값을 바꾸려면 프로그람은 lseek()체계호출을 진행하여야 한다. 파일을 열 때 핵심부는 파일지적자를 파일의 첫번째 바이트위치로 설정한다.

newoffset = lseek(fd, offset, whence);

매개 변수의 의미는 다음과 같다.

fd - 열린 파일의 파일서술자를 나타낸다.

offset - 파일지적자의 새로운 위치를 계산하는데 시용하는 옹근수값(정수 혹은 부수)을 지정한다.

whence - offset 값을 더하는 기준을 지정한다. 이 값에 따라 기준이 0(파일의 시작)이나 현재 파일지적자, 마지막 바이트의 위치(파일의 끝)가 된다.

read()체계호출의 매개 변수는 다음과 같다.

nread = read(fd, buf, count);

각 매개 변수의 의미는 다음과 같다.

fd - 열린 파일의 파일서술자를 나타낸다.

buf - 프로쎄스주소공간에 있는 자료를 전송할 완충기의 주소를 지정한다.

count - 읽어들일 바이트수를 나타낸다. 이 체계호출을 받으면 핵심부는 파일서술자fd가 가리키는 파일에서 열린 파일의 offset마당값인 현재 위치부터 시작하여 count 바이트만큼 읽으려고 한다. 파일의 끝에 도달하거나 비여있는 경우에는 count 바이트만큼 읽지 못할수도 있다. 여기서 되돌리는 nread값은 실제로 읽은 바이트수이다. 파일지적자는 이전값에 nread를 더한 값으로 갱신된다. write()에 전달히는 매개 변수도 이와 비슷하다.

파일닫기

프로쎄스가 더는 파일내용에 접근할 필요가 없는 경우 다음의 체계호출을 리용하여 파일을 닫을수 있다.

res = close(fd);

이 체계호출은 파일서술자 fd에 해당하는 열린 파일객체를 해제한다. 프로쎄스가 완료할 때 핵심부는 해당 프로쎄스가 열어놓은 모든 파일을 닫는다.

파일명변경과 파일삭제

파일명을 비꾸거나 파일을 지울 때에는 파일을 열 필요가 없다. 이런 작업은 대상파

일의 내용이 아닌 해당 등록부의 내용에 작용하기때문이다. 실례로 다음 체계호출은 파일련결의 이름을 바꾼다.

```
res = rename(oldpath, newpath);
```

또한 다음의 체계호출은 파일의 련결개수를 줄이고 해당 등록부입구점을 제거한다.

```
res = unlink(pathname);
```

파일은 련결개수가 0이 되여야만 삭제된다.

3. Linux핵심부의 개요

Linux핵심부는 응용프로그람이 동작할수 있는 실행환경을 제공한다. 따라서 핵심부는 일부 봉사와 이에 해당하는 대면부를 실현해야 한다. 응용프로그람은 이러한 대면부를 사용하며 대체로 하드웨어자원과 직접 호상작용하지는 않는다.

1) 프로쎄스와 핵심부모형

CPU는 사용자방식이나 핵심부방식중 어느 하나에서 동작한다. 그러나 실제로는 실행상태를 두개이상 지원하는 CPU도 있다. 실례로 인텔 80x86극소형처리소자는 서로 다른 실행상태 4가지를 지원한다. 그러나 모든 표준 Linux핵심부는 핵심부방식과 사용자방식만 사용한다.

사용자방식에서 동작중일 때 프로그람은 핵심부자료구조나 핵심부프로그람에 직접 접근할수 없다. 그러나 응용프로그람이 핵심부방식에서 동작중일 때에는 이러한 제한이 더는 적용되지 않는다. 개개의 CPU모형은 사용자방식에서 핵심부방식으로 절환하거나 이와 반대로 절환하는 특별한 명령을 제공한다. 프로그람은 대부분의 시간을 사용자방식에서 보내고 핵심부가 제공하는 봉사를 리용할 때에만 핵심부방식으로 절환한다. 핵심부는 프로그람의 요구를 처리한 후 프로그람을 다시 사용자방식으로 돌려보낸다. 프로쎄스는 체계에서 수명이 제한되여있는 동적인 개체이다. 핵심부에 있는 여러 루틴(routine)이 프로쎄스생성과 제거, 동기화하는 일을 한다.

핵심부는 프로쎄스가 아니라 프로쎄스관리자로 된다. 프로쎄스/핵심부모형에서는 프로쎄스가 핵심부봉사를 받으려고 할 때 《체계호출(system call)》이라는 특별한 프로그람구조를 리용하도록 한다. 각 체계호출은 프로쎄스가 제기하는 요청을 나타내는 매개 변수그룹을 설정한 후 사용자방식에서 핵심부방식으로 절환하는 하드웨어 고유의 CPU명령을 실행한다.

Linux체계에는 사용자프로쎄스외에 다음과 같은 특징을 가지는 《핵심부스레드(kernel thread)》라는 몇가지 특권프로쎄스가 있다.

- 핵심부주소공간에서 핵심부방식으로 실행한다.
- 사용자와 호상작용하지 않는다. 따라서 말단장치가 필요없다.
- 일반적으로 체계를 시동할 때 만들어지고 체계가 끝날 때까지 살아있다.

단일처리장치체계에서는 한번에 단 하나의 프로쎄스만 실행할수 있다. 프로쎄스는

사용자방식이나 핵심부방식중 하나에서 동작한다. 만약 핵심부방식에서 동작중이라면 처리장치는 어떤 핵심부루틴을 실행하고있을것이다. 그림 1-2는 사용자방식과 핵심부방식사이의 절환을 보여준다. 사용자방식에 있는 프로쎄스 1이 체계호출을 진행하면 프로쎄스를 핵심부방식으로 절환한 후 체계호출을 처리한다. 프로쎄스1은 사용자방식에서 실행을 재개하고 시계새치기(timer interrupt)가 발생하면 핵심부방식에서 순서짜기프로그람이 활성화된다. 다음 프로쎄스절환이 일어나 프로쎄스2가 사용자방식에서 실행을 시작하며 하드웨어장치에서 새치기가 발생할 때까지 계속 실행한다. 새치기가 발생하면 프로쎄스 2는 핵심부방식으로 절환하여 새치기를 처리한다.

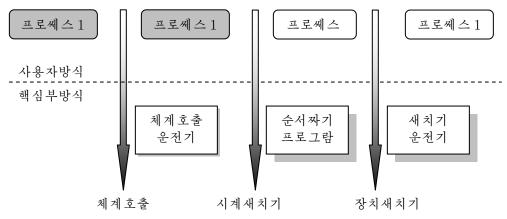



그림 1-2. 사용자방식과 핵심부방식사이의 전환

Linux핵심부는 체계호출외에도 많은 일을 수행한다. 사실 핵심부루틴이 호출되는 경우는 여러가지이다.

- 프로쎄스가 체계호출을 한 경우
- 프로쎄스를 실행하던 CPU가 《레외(exception)》를 발생시킨 경우
  레외라는것은 잘못된 명령의 실행처럼 비정상적인 상태를 말한다. 핵심부는 레외가 발생한 프로쎄스를 대신하여 레외를 처리한다.
- 주변장치(peripheral device)는 주목을 끌 필요가 있거나 상태변경, 입출력동작완료 같은 사건을 알려주기 위해 CPU에 《새치기신호(interrupt signal)》를 보낸다. 각 새치기신호는 《새치기처리기(interrupt handler)》라는 핵심부프로그람이 처리한다. 주변장치는 CPU와 비동기적으로 동작하므로 새치기가 언제 발생할지는 예측할수 없다.
- 핵심부스레드를 실행한 경우
  핵심부스레드는 핵심부방식에서 동작하므로 해당 프로그람은 핵심부의 일부로 간주해야 한다.

2) 프로쎄스실현

핵심부는 프로쎄스를 조종하기 위해서 매개 프로쎄스를 프로쎄스의 현재 상태정보를 포함하는 《프로쎄스서술자(process descriptor)》로 나타낸다.

핵심부는 프로쎄스의 실행을 중단할 때 다음과 같은 처리장치등록기들의 현재 내용을 프로쎄스서술자에 보관한다.

- 프로그람계수기(PC: program counter)와 탄창지시기(SP: stack pointer) 등록기
- 일반(general purpose)등록기
- 류동소수점(floating point)등록기
- CPU상태정보를 담고있는 처리장치조종등록기[처리장치상태단어(Processor Status Word)]
- 프로쎄스가 접근하는 주기억기를 관리하는데 사용하는 기억기관리등록기

핵심부가 어떤 프로쎄스를 다시 실행하려고 한다면 적절한 프로쎄스서술자마당을 CPU등록기로 읽어들인다. 보존된 프로그람계수기의 값은 마지막에 실행한 명령의 다음 명령을 가리키므로 프로쎄스는 이전에 멈춘 위치부터 실행을 재개한다.

프로쎄스가 CPU에서 실행중이 아닐 때에는 어떤 사건이 일어나기를 기다린다. Linux핵심부는 대기상태를 여러 가지로 구별하며 대체로 프로쎄스서술자대기렬(queue)을 리용하여 이것을 구현한다. 각 대기렬은(대기렬은 비여있을수도 있다.) 특별한 사건을 기다리는 프로쎄스의 집합이다.

3) 재진입가능한 핵심부

Linux핵심부는 《재진입(reentrant)》가능하다.이 말은 여러 프로쎄스를 동시에 핵심부방식에서 실행할수 있다는것을 의미한다. 물론 단일처리장치체계에서는 한 프로쎄스만 작업을 진행할수 있지만 여러 프로쎄스가 CPU나 어떤 입출력작업이 완료되기를 기다리며 핵심부방식에서 차단된 상태로 있을수 있다. 실례로 어떤 프로쎄스의 요구에 따라 디스크를 읽으라는 요청을 하면 핵심부는 디스크조종기(disk controller)에 이것을 처리하도록 지시한 후 다른 프로쎄스를 실행한다. 장치가 새치기를 발생시켜 핵심부에 읽기작업이 끝났다고 알려주면 이전 프로쎄스는 실행을 재개할수 있게 된다.

재진입을 가능하게 하는 방법 중 하나는 국부변수만 수정하고 대역자료구조는 수정하지 않도록 함수를 작성하는것이다. 이런 함수를 재진입가능한 함수(reentrant function)라고 한다. 그러나 이런 재진입가능한 함수만으로 재진입가능한 핵심부를 구성할수 있는것은 아니다.(실제로 일부 실시간핵심부는 이렇게 구현되여있다.) 대신 핵심부가 재진입할수 없는 함수를 포함하여 어떤 시점에서 오직 한 프로쎄스만 재진입할수 없는 함수를 실행하도록 하기 위해 잠그기(locking)기구를 리용한다. 모든 프로쎄스는 핵심부방식에서도 자기에게 할당된 기억기에서만 동작하며 다른 프로쎄스의 기억기를 간섭할수 없다.

하드웨어새치기가 발생하면 재진입가능한 핵심부는 현재 실행중인 프로쎄스가 비록

핵심부방식에 있더라도 현재프로세스를 잠시 보류할수 있다. 이 기능은 새치기를 발생시키는 장치조종기의 작업처리량을 증가시키기때문에 매우 중요하다. 장치는 새치기를 발생시킨 후 CPU가 이것을 인식할 때까지 기다린다. 이때 핵심부가 새치기에 빠르게 응답한다면 장치조종기는 CPU가 새치기를 처리하는 동안 다른 작업을 수행할수 있을것이다.

이제 핵심부재진입과 이것이 핵심부구조에 미치는 영향을 살펴보자.《핵심부조종경로(kernel control path)》는 체계호출이나 례외, 새치기를 처리하면서 핵심부가 실행해나가는 명령의 순서를 말한다.

가장 간단한 경우 CPU는 처음 명령부터 마지막 명령까지 순차적으로 핵심부조종경로를 실행한다. 그렇지만 다음의 사건가운데서 하나라도 발생하면 CPU는 핵심부조종경로중간에 다른 핵심부조종경로를 끼워넣는다.

- 사용자방식에서 실행중인 프로쎄스가 체계호출을 했는데 해당 핵심부조종경로에서 프로쎄스가 요청한 작업을 바로 처리할수 없을수도 있다. 이때 핵심부는 순서짜기프로그람을 실행하여 새롭게 실행할 프로쎄스를 선택하며 그 결과 프로쎄스절환이 일어난다. 이때 첫번째 핵심부조종경로는 끝나지 않은 상태로 남고 CPU는 다른 핵심부조종경로를 실행한다. 이 경우 두 조종경로는 서로 다른 두 프로쎄스를 대신하여 실행된다. 핵심부조종경로를 실행하던중에 CPU가 례외를 일으킬 때, 실례로 주기억기에 존재하지 않는 페지에 접근하려고 한 경우 이때 첫번째 조종경로를 잠시 보류하고 CPU는 례외를 처리하기 위해 적절한 작업을 수행한다. 이를테면 프로쎄스용으로 새 페지를 할당하고 디스크에서 내용을 읽어들여 례외를 처리한다고 하자. 이 작업이 끝나면 첫번째 조종경로는 작업을 재개할수 있다. 이 경우 두 핵심부조종경로는 똑같은 프로쎄스를 대신하여 실행된다.
- 새치기를 허용한 상태에서 CPU가 핵심부조종경로를 실행하고있을 때 하드웨어새치기가 발생한 경우. 첫번째 핵심부조종경로가 완료하지 않은 상태에서 CPU는 새치기를 처리하기 위해 또 다른 핵심부조종경로를 시작한다. 새치기처리기가 완료하면 첫번째 핵심부조종경로는 작업을 재개한다. 이 경우 두 핵심부조종경로는 같은 프로쎄스의 실행결합부에서 실행되며 이 과정에서 소비된 체계시긴은 이 프로쎄스가 시용한것으로 계산된다. 그러나 새치기처리기가 해당 프로쎄스를 위해 동작할 필요는 없다.

그림 1-3는 핵심부조종경로를 처리하는 도중 다른 조종경로가 끼여 들지 않을 때와 끼여들 때 모두를 그림으로 보여준다. 그림에서는 세가지 CPU 상태가 있다.

- 사용자방식에서 프로쎄스 실행하기(사용자)
- 례외나 체계호출운전기 실행하기(례외)
- 새치기처리기 실행하기(새치기)

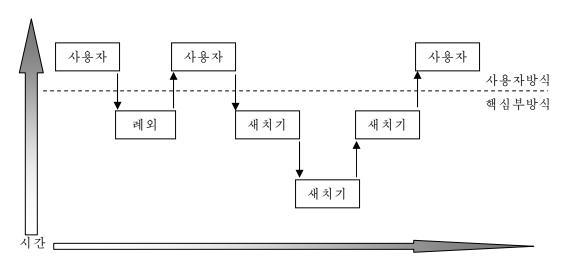



그림 1-3. 핵심부조종의 겹침

4) 프로쎄스주소공간

각 프로쎄스는 자기의 개인주소공간(private address space)에서 실행된다. 사용자방식에서 실행하는 프로쎄스는 자기의 탄창, 자료 그리고 코드령역을 사용한다. 한편 핵심부방식에서 실행할 때에는 핵심부자료와 코드령역에 접근하고 별도의 탄창을 사용한다.

핵심부는 재진입이 가능하므로 여러 핵심부조종경로(각각 서로 다른 프로쎄스와 련관되여있다.)가 차례로 실행될수 있다. 이 경우 각 핵심부조종경로는 자기의 핵심부탄창을 사용한다.

각 프로쎄스가 개인주소공간에 접근하는것처럼 보이지만 주소공간의 일부를 프로쎄스사이에서 공유하는 경우도 있다. 프로쎄스가 직접 주소공간을 공유할것을 요구하는 경우도 있고 핵심부가 기억기사용량을 줄이려고 자동으로 공유하게 만들기도 한다.

똑같은 프로그람, 실례로 편집기를 여러 사용자가 동시에 사용하면 프로그람은 기억기에 한번만 적재되고 이것을 사용하는 모든 사용자가 명령을 공유한다. 물론 각 사용자마다 서로 다른 자료가 있으므로 자료를 공유하면 안된다. 핵심부는 기억기를 아끼려고 이렇게 자동으로 주소공간을 공유하게 한다.

또한 프로쎄스는 프로쎄스사이통신기구의 한 종류로서 SystemV에서 도입하고 Linux가 지원하는 《공유기억기(shared memory)》기법을 통하여 자신의 주소공간의 일부를 공유할수 있다.

마지막으로 Linux는 mmap()체계호출을 지원한다. 이 체계호출은 파일이나 장치에 있는 기억기의 일부를 프로쎄스주소공간의 일부로 배치한다. 기억기배치는 자료를 전송하는 일반적인 방법인 읽기와 쓰기에 대한 대안으로 된다. 똑같은 파일을 여러 프로쎄스

가 공유하면 이것을 공유하는 각 프로쎄스의 주소공간에 해당 파일의 기억기배치가 들어간다.

5) 동기화와 림계령역

재진입 가능한 핵심부를 실현하려면 동기화(synchronization)를 리용해야 한다. 핵심부조종경로가 어떤 핵심부자료구조를 가지고 작업하던 도중에 멈춘 경우 이 자료구조가 옳바른 상태로 다시 설정되지 않는한 다른 핵심부조종경로가 똑같은 자료구조를 사용할수 없게 해야 한다. 그렇지 않으면 두 조종경로의 호상작용으로 인해 자료구조에 보관된 정보가 엉망이 될수도 있다.

실례를 들어 어떤 체계자원을 사용할수 있는 개수를 나타내는 대역변수 v가 있다고 하자. 첫번째 핵심부조종경로 A가 이 값을 읽고 사용가능한 항목이 단 하나인것을 알아낸다. 이 시점에서 다른 핵심부조종경로 B가 활성화되여 똑같은 변수를 읽으면 여전히 값은 1일것이다. 그래서 B가 v를 감소시키고 자원을 사용하기 시작한다고 하자. 그리고나서 A가 실행을 재개하면 A는 이미 v의 값을 읽었으므로 v 값을 감소시키고 자원을 사용할수 있다고 여기는데 이미 B가 해당 자원을 사용하는 중이다. 최종결과로 v는 -1이 되고 두 핵심부조종경로는 똑같은 자원을 사용하게 되여 잠재적으로 심각한 결과를 초래할수도 있다.

둘이상의 프로쎄스를 어떻게 순서짜기하는가에 따라 계산결과가 달라질수 있다면 해당 코드는 질못된것이다. 이 경우 이 코드에 《경쟁조건(race condition)》이 있다고 한다.

일반적으로 《원자적인 연산(atomic operation)》을 리용하면 대역변수에 안전하게 접근할수 있다. 앞의 실례에서 두 조종경로가 중단할수 없는 단일연산을 사용하여 v값을 읽고 감소시킨다면 자료가 잘못되는 일은 발생하지 않을것이다. 그러나 핵심부에는 이런 단일연산으로 다룰수 없는 많은 자료구조가 있다. 실례를 들어 련결목록에 있는 항목을 제거하는 경우 적어도 두개의 지적자에 동시에 접근해야 하므로 단일연산으로 이것을 수행할수 없다. 어떤 코트령역이든 일단 시작했다면 다른 프로쎄스가 해당령역에 진입하기 전에 끝내야만 하는 령역을 《림계령역(critical region)》이라고 한다.

이런 문제는 핵심부조종경로사이에서뿐만 아니라 자료를 공유하는 프로쎄스사이에서도 일어난다. 이것을 해결하기 위해 몇가지 동기화기법을 받아들였는데 앞으로 나가면서 보기로 한다.

6) 비선취형핵심부

대부분의 고전Unix는 핵심부를 비선취형(nonpreemptive)으로 만들어 동기화문제를 아주 간단히 해결하였다. 비선취형핵심부에서는 프로쎄스가 핵심부방식에서 실행중이면 이것을 임의로 보류하거나 다른 프로쎄스로 교체할수 없다. 따라서 단일처리장치체계에서 핵심부는 새치기처리기나 례외처리기에서 갱신하지 않는 모든 핵심부자료구조에 안전하게 접근할수 있다.

물론 핵심부방식에 있는 프로쎄스가 자발적으로 CPU를 반납할수 있는데 이 경우 CPU를 반납하기 전에 모든 자료구조가 온전한 상태에 있도록 해야 한다. 게다가 프로쎄스는 실행을 재개할 때 이전에 접근한 자료구조가 바뀌였을수도 있으므로 다시 검사해야 한다. 다중처리장치체계에서는 서로 다른 CPU에서 실행되는 두 핵심부조종경로가 동시에 똑같은 자료구조에 접근할수도 있다. 따라서 다중처리장치체계에서는 비선취성이 효률적인것이 못된다.

7) 새치기금지

단일처리장치체계를 위한 다른 동기화기구로서 림계령역에 들어가기 직전에 모든 하드웨어새치기를 금지하고 림계령역을 빠져나오는 즉시 다시 허용하는 방법이 있다. 이 기구는 간단하지만 최적의 방법은 아니다. 림계령역이 크면 새치기는 상대적으로 긴 시간동안 금지된 상태로 남아있어 자칫하면 모든 하드웨어의 동작을 멈추게 할수 있기때문이다. 더구나 다중처리장치체계에서는 이런 기구가 전혀 동작하지 않는다. 다중처리장치체계에서는 림계령역에서 사용하는 똑같은 자료구조를 다른 CPU에서 접근하지 못하도록 할수 있는 방법이 없다.

8) 신호기

단일처리장치체계와 다중처리장치체계에서는 다같이 《신호기(semaphore)》라고 하는 효률적인 기구를 많이 사용하고있다. 신호기는 어떤 자료구조와 련관된 간단한 계수기이며 모든 핵심부스레드는 자료구조에 접근하기 전에 해당 신호기를 검사한다. 각 신호기는 다음과 같은 항목으로 구성된 객체이다.

- 옹근수변수
- 대기하는 프로쎄스목록
- 두개의 원자적인 메쏘드(method): down()과 up()

down()메쏘드는 신호기의 값을 감소시킨다. 결과 값이 0보다 작으면 이 메쏘드는 프로쎄스를 신호기목록에 추가하고 차단한다.(즉 순서짜기프로그람을 호출한다.) up()메쏘드는 신호기의 값을 증가시키고 결과값이 0보다 작거나 같으면 신호기목록에 있는 하나이상의 프로쎄스를 다시 활성화한다.

보호를 받는 각 자료구조는 자기의 신호기를 가지며 초기값은 1이다. 핵심부조종경로가 자료구조에 접근하려고 할 때에는 먼저 해당 신호기에 down()메쏘드를 실행한다. 새로운 신호기의 값이 부수가 아니면 해당 자료구조에 접근할수 있고 그렇지 않으면 핵심부조종경로를 실행하던 프로쎄스를 신호기목록에 추가하고 차단한다. 다른 프로쎄스가 이 신호기에 up()메쏘드를 실행하면 신호기목록에 있는 프로쎄스중 하나는 작업을 계속할수 있다.

9) 스핀잠그기

다중처리소자체계에서는 신호기가 동기화문제를 해결하는 가장 좋은 방법은 아니다. 어떤 핵심부자료구조는 서로 다른 CPU에서 동작하는 핵심부조종경로가 동시에 접근하

지 못하게 해야 한다. 이때 자료구조를 갱신하는데 필요한 시간이 짧으면 신호기는 매우 효률이 떨어진다. 신호기를 검사하여 신호기가 사용중이면 핵심부는 프로쎄스를 신호기 목록에 삽입한 후에 프로쎄스를 보류시켜야 한다. 이 두 작업은 상대적으로 비용(즉 시간)이 많이 들기때문에 작업을 마쳤을 때 다른 조종경로에서 이미 해당 신호기를 반납했을수도 있다.

이런 경우 다중처리장치조작체계는 《스핀잠그기(spin lock)》를 리용한다. 스핀잠그기는 신호기와 매우 류사하지만 프로쎄스목록이 크다는 점이 다르다. 프로쎄스가 시용하려는 잠그기를 다른 프로쎄스가 이미 사용하고있다면 프로쎄스는 잠그기가 풀릴 때까지 반복해서 명령을 실행하며 《회전(spin)》한다. 단일처리장치환경에서는 스핀잠그기가 필요없다. 단일처리장치환경에서는 핵심부조종경로가 잠겨있는 자료구조에 접근하려고 하면 무한순환을 시작하기때문에 해당 자료구조를 보호하며 갱신하던 핵심부조종경로는 실행을 계속할수도 없고 스핀잠그기를 반납할수도 없다. 그 결과 체계가 죽게 된다.

10) 교착회피

다른 조종경로와 동기화해야 하는 프로쎄스나 핵심부조종경로는 쉽게 《교착(deadlock)》상태에 빠져들수 있다. 교착이 발생하는 가장 단순한 경우는 프로쎄스 p1이 자료구조 a에 대한 접근권한을 가지고있고 프로쎄스 p2가 b에 대한 접근권한을 기지고있는데 p1이 b를 기다리고 p2가 a를 기다릴 때이다. 여러 프로쎄스그룹사이에서 더 복잡한 순환대기가 발생할수도 있다. 응당 교착상태는 이에 영향을 받는 프로쎄스와 핵심부조종경로를 완전히 정지시키는 결과를 가져온다.

핵심부설계관점에서 보면 교착은 사용하는 핵심부신호기의 수가 많을 때 문제가 된다. 이런 경우 핵심부조종경로중간에 삽입되는 모든 경우에 교착상태가 절대로 일어나지 않도록 하기는 매우 어렵다. Linux를 포함한 여러 조작체계는 제한된 수의 신호기를 사용하며 신호기에 대한 요청을 오름순서(ascending order)로 정렬하여 이 문제를 피한다. 앞의 실례에서 두 프로쎄스가 신호기에 대한 요청을 a, b 순서로 하기로 약속한다면 서로 상대방이 신호기를 풀어주기를 기다리는 일은 발생하지 않는다.

11) 신호와 프로쎄스사이통신

Linux《신호(signal)》는 프로쎄스에 체계사건을 알려주는 기구를 제공한다. 각 사건은 SIGTERM과 같이 기호로 된 상수로서 참조하는 자체의 신호번호를 가지고있다. 체계사건에는 두 종류가 있다.

비동기적알림(asynchronous notification)

실례로 사용자가 말단에서 새치기건을 눌러서(보통은 Ctrl+C) 전경프로쎄스(foreground process)에 SIGINT새치기신호를 보낼수 있다.

동기적오유(synchronous error)**와 례외**(exception)

실례로 프로쎄스가 잘못된 주소에 있는 기억위치에 접근하려고 하면 핵심부는 프로쎄스에 SIGSEGV신호를 보낸다. POSIX 표준에서는 대략 20개의 신호를 정의하고있다.

그 가운데서 2개는 사용자가 정의할수 있으며 사용자방식에 있는 프로쎄스사이에서 통신과 동기화를 하기 위한 원시적인 기구로서 이 신호를 리용할수 있다. 프로쎄스는 신호를 받을 때 보통 다음 두가지 반응을 보인다.

- 신호를 무시한다.
- 비동기적으로 특별한 절차(신호취급기)를 실행한다.
- 프로쎄스가 그중 하나를 지정하지 않으면 핵심부는 신호번호에 따라 정해진 기정동작(default action)을 수행한다. 기본동작은 다음 다섯가지이다.
- 프로쎄스를 완료한다.
- 실행결합부와 주소공간의 내용을 파일에 기록하고(핵심정보(core dump)) 프로쎄스를 완료한다.
- 신호를 무시한다.
- 프로쎄스를 보류한다.
- 프로쎄스가 중단된 상태라면 프로쎄스실행을 재개한다.

POSIX에서는 프로쎄스가 일시적으로 신호를 차단할수 있도록 규정하기때문에 핵심부에서 신호처리는 어느 정도 복잡하다. 더구나 SIGKILL과 SIGSTOP신호는 프로쎄스가 직접 처리할수 없으며 무시할수도 없다.

AT&T의 UNIX System V는 사용자방식에서 사용할수 있는 다른 종류의 프로쎄스사이통신(interprocess communication) 방법을 도입했으며 여러 Unix핵심부에서 이것을 도입하였다. 이러한 신호기(semaphore), 통보문대기렬(message queue), 공유기억기(shared memory)를 한데 묶어서 System V IPC라고 한다. 핵심부는 이것을 《IPC자원》을 통해 실현한다. 프로쎄스는 shmget(), semget(), msgget() 체계호출을 통하여 자원을 획득한다. 파일과 마찬가지로 IPC자원도 기억기에 계속 존재하므로 이것을 만든 프로쎄스나 현재 소유자, 관리자프로쎄스에서 명백하게 할당을 해제해야 한다.

신호기는 앞에서의 《동기화와 림계령역》에서 설명한 신호기와 비슷하지만 사용자방식프로쎄스를 위한것이라는 점이 다르다. 통보문대기렬은 프로쎄스가 msgsnd(), msgget()체계호출을 리용하여 통보문을 교환할수 있게 한다. 이 함수들은 각각 지정된 통보문대기렬에 통보문을 넣고 뽑아낸다.

공유기억기는 프로쎄스사이에서 자료를 교환하고 공유하는 가장 빠른 방법이다. 프로쎄스는 shmget()체계호출을 통하여 필요한 크기만큼 공유기억기를 생성한다. IPC자원ID를 얻으면 프로쎄스는 shmat()체계호출을 통하여 자기의 주소공간에 있는 새로운 령역의 시작주소를 얻는다. 프로쎄스가 공유기억기를 자기의 주소공간에서 떼여낼 때에는 shmdt()체계호출을 진행한다. 공유기억기실현은 핵심부가 프로쎄스주소공간을 어떻게 실현하는가에 따라 달라진다.

12) 프로쎄스관리

Linux는 프로쎄스와 프로쎄스가 실행하는 프로그람을 명확히 구별한다. 그렇기때

문에 새로운 프로쎄스를 생성하고 완료하는데는 각각 fork()와 _exit() 체계호출을 사용하는 반면에 새로운 프로그람을 적재하는데는 exec()계렬의 체계호출을 사용한다. exec()계렬의 체계호출을 실행하면 프로쎄스는 적재한 프로그람을 포함한 새로운 주소공간에서 실행을 재개한다. fork()를 호출하는 프로쎄스는 《부모(parent)》, 새로운 프로쎄스는 《자식(child)》이 된다. 프로쎄스를 서술하는 자료구조에는 친부모와 모든 친자식을 가리키는 지적자가 들어있으므로 부모와 자식은 서로를 알수 있다.

fork()함수를 단순하게 실현하려면 부모의 코드와 자료를 자식에게 그대로 복제해주면 된다. 그러나 이 방법은 시간이 많이 걸린다. 그래서 최신핵심부들은 하드웨어의 페지화장치(paging unit)기능을 활용하여 《쓰기복사(Copy on Write)》한다. 이것은 페지복제를 최후순간(즉 부모나 자식이 페지에 쓰려고 할 때)까지 미루는것이다.

_exit()체계호출은 프로쎄스를 완료한다. 핵심부는 프로쎄스가 점유하고있는 자원을 반납하고 부모프로쎄스에 SIGCHLD신호를 보내는데 대체로 부모는 이 신호를 무시한다.

13) 좀비프로쎄스

그럼 부모프로쎄스는 자식프로쎄스가 완료했는지 어떻게 알수 있겠는가? 부모프로쎄스는 wait()체계호출을 통하여 자식프로쎄스가운데서 하나가 완료할 때까지 기다릴수 있다. 이 체계호출은 완료한 자식의 PID(Process ID)를 돌려준다.

이 체계호출을 진행하면 핵심부는 이미 완료한 자식이 있는지 검사한다. 여기서 완료한 프로쎄스를 나타내는 《좀비프로쎄스(zombie process)》상태라는 독특한 상태가 등장한다. 완료한 프로쎄스는 부모프로쎄스가 자기에게 wait()체계호출을 진행하기 전까지 이 상태로 남는다. 이 체계호출운전기는 프로쎄스서술자(descriptor)에서 자원사용에 관한 자료를 추출하는데 이 자료를 얻어온 후에야 프로쎄스서술자를 해제할수 있다. wait()체계호출을 진행할 때 그전에 완료한 자식프로쎄스가 없으면 핵심부는 보통 자식프로쎄스가 완료할 때까지 프로쎄스를 대기상태로 만든다.

많은 핵심부가 waitpid()체계호출도 실현하고있다. 이 체계호출은 특정한 자식프로쎄스가 완료할 때까지 기다리게 한다. 이 밖에도 다른 wait()계렬의 체계호출도 있다. 부모프로쎄스가 wait()체계호출을 진행할 때까지 자식프로쎄스의 정보를 그대로 가지고 있는것은 좋지만 만약 부모프로쎄스가 이 체계호출을 진행하지 않고 완료했다면 어떻게 될것인가? 실행을 마친 자식프로쎄스의 정보는 살아있는 다른 프로쎄스가 사용할수도 있는 유용한 기억기를 그대로 점유할것이다. 실례로 많은 쉘(shell)들이 사용자가 명령을 배경(background)작업으로 실행한 다음 가입탈퇴(log out)하는것을 허용한다. 이때 명령을 실행한 쉘은 완료하지만 자식프로쎄스는 실행을 계속한다.

이런 상태는 체계초기화과정에 만드는 init이라는 특별한 체계프로쎄스로 해결한다. 프로쎄스가 완료하면 핵심부는 해당 프로쎄스의 모든 자식프로쎄스의 프로쎄스서술자에 있는 지적자를 바꿔서 이것들을 init프로쎄스의 자식으로 만든다. init프로쎄스는 모든 자식의 실행상태를 지켜보고 정기적으로 wait()체계호출을 진행한다. 이것은 모든 좀비

프로쎄스를 제거할수 있게 한다.

14) 프로쎄스그룹과 가입세션

최근의 Unix조작체계는 작업(job)이라고 하는 추상적인 개념을 나타내기 위해 《프로쎄스그룹(process group)》이라는 개념을 도입한다. 실례로 다음과 같은 명령을 실행하면 배시(bash)와 같이 프로쎄스그룹을 지원하는 쉘은 ls, sort, more 3개의 프로쎄스를 위해 새로운 프로쎄스그룹을 생성한다.

$ ls | sort | more

이렇게 해서 쉘은 3개의 프로쎄스가 마치 하나(정확히 말하면 한 작업)인것처럼 여기게 한다. 각 프로쎄스서술자에는 프로쎄스그룹 ID마당이 있다. 각 프로쎄스그룹에는 그룹우두머리(group leader)가 있을수 있는데 그룹우두머리란 PID가 프로쎄스그룹ID와 일치하는 프로쎄스를 말한다. 새로 만들어진 프로쎄스는 처음에 부모의 프로쎄스그룹으로 들어간다. 최근의 Unix핵심부는 《가입세션(login session)》이라는 개념도 새로 도입하였다. 비공식적으로 가입세션은 특정말단에서 작업세션(working session)을 시작한 프로쎄스(보통은 사용자를 위해 만들어진 첫번째 명령 쉘프로쎄스)의 모든 자식프로쎄스를 포함한다. 같은 프로쎄스그룹에 있는 모든 프로쎄스는 동일한 가입세션에 있어야 한다. 가입세션에서는 여러 프로쎄스그룹이 동시에 활동하고있을수 있다. 이 프로쎄스그룹가운데서 하나는 항상 전경에 있으며 말단에 접근할수 있다. 다른 활동중인 프로쎄스그룹은 배경에 있다. 배경프로쎄스가 말단에 접근하려고 하면 해당 프로쎄스는 SIGTTIN이나 SIGTTOUT신호를 받는다. 많은 명령쉘에서 내부명령 bg와 fg를 리용해서 프로쎄스그룹을 배경이나 전경에 둘수 있다.

15) 기억기관리

기억기관리는 Linux핵심부에서 가장 복잡하게 동작하는 부분이다.

모든 최신Unix체계는 《가상기억기(virtual memory)》라는 추상화개념을 제공하고있다. 가상기억기는 응용프로그람에서 요구하는 기억기와 하드웨어기억관리장치(MMU, Memory Management Unit) 사이에서 론리적인 계층처럼 동작한다. 가상기억기를 사용하면 여러 측면에서 우점이 있다.

- 여러 프로쎄스를 동시에 실행할수 있다.
- 사용할수 있는 물리기억기보다 많은 기억기를 요구하는 응용프로그람을 실행할수 있다.
- 프로그람코드가운데서 일부만 기억기에 적재해도 프로쎄스를 실행할수 있다.
- 각 프로쎄스는 사용가능한 물리기억기의 일부에만 접근할수 있다.
- 서고나 프로그람의 기억기영상 하나를 프로쎄스사이에서 공유할수 있다.
- 프로그람을 재배치(relocation)할수 있다. 즉 물리기억기 어디에나 둘수 있다.
- 프로그람작성자는 물리기억기의 구조에 신경쓸 필요가 없으므로 기계에 의존하지 않는 코드를 작성할수 있다.

가상기억기보조체계(subsystem)의 핵심적인 개념은 가상주소공간(virtual address space)이다. 프로쎄스가 참조할수 있는 기억기주소의 집합은 물리기억기주소와 다르다. 프로쎄스가 가상주소를 사용하면 핵심부와 MMU가 서로 협력하여 요구한 기억주소의 실제의 물리적인 위치를 찾는다. 오늘날의 CPU에는 자동으로 가상주소를 물리주소로 변환하는 하드웨어회로가 있다. 이에 따라 사용가능한 기억기를 4kB나 8kB 단위의 페지틀(page frame)로 쪼개고 페지표(page table)를 통해 가상주소와 물리주소사이의 대응관계를 지정한다. 이 회로를 사용하면 련속된 가상주소의 기억기를 할당해달라는 요청을 물리주소가 련속되지 않는 페지틀그룹을 할당하여 처리할수 있어 기억기할당이 간단해진다.

기억기사용

모든 Unix조작체계는 주기억기(RAM, Random Access Memory)를 두개의 부분으로 나누어 구분한다. 몇Mbyte는 핵심부영상(즉 핵심부코드와 핵심부의 정적자료구조)을 보관하는데 사용한다. 나머지는 보통 가상기억체계가 관리하는 부분으로서 다음의 세가지 목적으로 사용한다.

- 핵심부에서 요구되는 완충기(buffer)와 서술자(descriptor), 기타 동적으로 만들어지는 핵심부자료구조용으로 쓰인다.
- 프로쎄스에게 필요한 일반적인 기억기령역과 파일을 기억기에 배치하는데 쓰인다.
- 디스크나 완충기를 비롯하여 다른 장치로부터 더 좋은 성능을 얻기 위한 완충기억기로 쓰인다.

각이한 형태의 기억기요구에 대하여 사용할수 있는 기억기는 제한되여있기때문에 이러한 요구사이에 균형을 맞추어야 한다. 이것은 특히 사용가능한 기억기가 거의 남아있지 않을 때 더욱 중요하게 제기된다. 사용가능한 기억기가 림계값에 이르면 여분의 기억기를 추가로 만들기 위해 페지틀을 해방하는 알고리듬을 호출한다. 이때 과연 어떤 페지틀을 해방하는것이 가장 적합하겠는가? 이 질문에는 간단히 대답할수 없으며 리론적으로도 내용이 심오하다. 유일한 방법은 경험적으로 알고리듬을 조정해나가며 조심스럽게 개발하는것이다.

가상기억체계가 반드시 해결하고 넘어가야 하는 중요한 문제들가운데서 하나는 《기억기쪼각화(memory fragmentation)》이다. 리론적으로는 사용할수 있는 페지틀의 수가 너무 적을 때에만 기억기요구가 실패한다. 그러나 핵심부는 때때로 물리적으로 련속인 기억령역을 사용해야만 하는 경우가 있다. 따라서 사용할수 있는 기억기는 충분하지만 련속된 블로크로 사용할수 있는 기억기의 크기가 작을 때에는 요청이 실패할수 있다.

핵심부기억기할당자

《핵심부기억기할당자(KMA, Kernel Memory Allocator)》는 체계의 모든 부분의 기억기령역관련요청을 처리하는 보조체계이다. 이 요청중 일부는 핵심부에서 사용할 기억기를 필요로 하는 다른 핵심부보조체계에서 오고 일부는 사용자프로그람이 자기의

프로쎄스주소공간을 늘이기 위해서 체계호출을 통해 오기도 한다. 좋은 핵심부기억기할당자는 다음과 같은 특징을 가지고있어야 한다.

- 핵심부기억기할당자는 빨라야 한다. 실제로 모든 핵심부보조체계에서(새치기처리기를 포함하여) 기억기를 요청하기때문에 이 점은 가장 중요하다.
- 랑비되는 기억기의 량을 최소로 하여야 한다.
- 기억기쪼각화를 될수록 줄여야 한다.
- 다른 기억기관리보조체계와 협력하여 그들의 기억기를 빌려오거나 그들이 가지고있는 기억기를 해방할수 있어야 한다.

지금까지 제안된 기본적인 핵심부기억기할당자의 알고리듬기법들을 보면 다음과 같다.

- 자원배치할당자(resource map allocator)
- 2의 제곱 자유목록(free list)
- 맥쿠식-카렐(McKusick-Karels)할당자
- 형제체계(buddy system)
- 마크(Mach)의 지역할당자(zone allocator)
- Dynix할당자
- Solaris의 스랩(Slab) 할당자

프로쎄스의 가상주소공간다루기

프로쎄스주소공간에는 프로쎄스가 접근할수 있는 모든 가상기억주소가 들어있다. 핵심부는 프로쎄스의 가상주소공간을 《기억기령역서술자(memory area descriptor)》의 목록에 보관한다. 실례로 프로쎄스가 exec()계렬의 체계호출을 하여 어떤 프로그람을 실행하면 핵심부는 프로쎄스에 다음과 같은 기억기령역으로 구성된 가상주소공간을 할당한다.

- 프로그람의 실행코드
- 프로그람의 초기화된 자료
- 프로그람의 초기화되지 않은 자료
- 초기프로그람탄창(즉 사용자방식탄창)
- 프로그람이 필요로 하는 공유서고(shared library)의 실행코드와 자료
- 동적기억기(heap, 프로그람이 동적으로 요구하는 기억기)

모든 최신 Unix조작체계는 《요구페지화(demand paging)》라는 기억기할당방책을 받아들이고있다. 요구페지화를 리용하면 물리기억기에 프로그람의 페지가 전혀 없더라도 프로쎄스는 프로그람실행을 시작할수 있다. 프로쎄스가 존재하지 않는 페지에 접근하면 기억기관리장치는 례외를 발생시키고 례외처리기는 해당 기억기령역을 찾아서 자유페지를 할당한 후 이것을 정확한 자료로 초기화한다. 이와 비슷하게 프로쎄스가 malloc()를 사용하거나 brk()체계호출(malloc()는 내부적으로 이 체계호출을 실행한다)을 호출하여 동적으로 기억기를 요구하면 핵심부는 해당 프로쎄스의 동적기억령역크

기만 갱신한다. 나중에 핵심부는 프로쎄스가 가상기억령역에 접근하다가 레외가 발생할 경우에만 페지틀을 할당한다. 가상주소공간은 앞에서 언급한 《Copy On Write》 같은 효률적인 방책을 가능하게 한다. 실례로 새로운 프로쎄스를 생성할 때 핵심부는 단순히 부모의 페지틀을 읽기전용으로 표시한 후 자식의 주소공간에 할당한다. 그래서 부모나 자식이 페지의 내용을 수정하려고 하는 즉시 레외가 발생하고 레외처리기는 내용을 수정하려고 한 프로쎄스에 새로운 페지틀을 할당하고 이것을 원래 페지의 내용으로 초기화한다.

바꾸기와 고속완충(caching)

Unix조작체계는 프로쎄스가 사용할수 있는 가상주소공간의 크기를 확장하기 위해 디스크에 있는 《바꾸기령역(swap area)》을 리용한다. 가상기억체계는 한 페지틀의 내용을 기본적인 바꾸기단위로 여긴다. 프로쎄스가 바꾸어내보내기(swap out)한 페지에 접근하려고 할 때마다 기억기관리장치는 레외를 발생시킨다. 그러면 레외처리기가 새로운 페지틀을 할당하고 페지틀의 내용을 디스크에 보관된 이전 내용으로 초기화한다.

한편 물리기억기를 하드디스크나 다른 블로크장치를 위한 고속완충기억기(cache)로도 사용할수 있다. 이것은 하드디스크구동장치가 매우 느리기때문에 디스크에 접근하는데는 종종 체계성능에서 문제가 된다. 초기에 Unix체계에서도 디스크에서 읽은 블로크에 대응하는 디스크완충기들을 주기억기에 줌으로써 디스크에 기록하는 작업을 될수록 뒤로 미루는 방책으로 되여있었다. sync()체계호출은 모든 《불결한(dirty)》완충기(즉 완충기의 내용과 이에 대응하는 디스크블로크의 내용이 다른 모든 완충기)를 디스크에 기록하여 디스크의 동기화를 강제로 진행한다. 모든 조작체계는 자료를 잃지 않도록 주기적으로 불결한 완충기를 디스크에 기록하는데 신경을 쓴다.

16) 장치구동기

핵심부는 《장치구동기(device driver)》를 통해 입출력장치와 호상작용한다. 장치구동기는 핵심부에 들어있으며 하드디스크나 건반, 마우스, 모니터, SCSI모선에 련결된 장치, 망카드와 같은 장치를 하나이상 조종하는 자료구조와 함수로 이루어진다. 각 구동기는 특정한 대면부에 따라 핵심부의 다른 부분과(다른 구동기와도) 호상작용한다.

이런 접근 방법에는 다음과 같은 우점이 있다.

- 장치에 따라 고유한 코드를 특정한 모듈안에 넣을수 있다.
- 제품생산자는 핵심부원천코드를 모르더라도 새로운 장치를 추가할수 있다. 단지 대면부명세(Interface specification)만 알면 된다 .
- 핵심부는 모든 장치를 똑같은 방법으로 다루고 똑같은 대면부로 접근한다.
- 체계를 재시동하지 않고도 핵심부에 동적으로 적재할수 있는 모듈형태로 장치구동기를 만들수 있다. 또한 더는 필요없는 모듈을 동적으로 부리워 핵심부영상이 주기억기에서 차지하는 크기를 줄일수 있다.

그림 1-4는 장치구동기가 핵심부의 다른 부분과 프로쎄스와 어떻게 호상작용하는가를 보여준다.

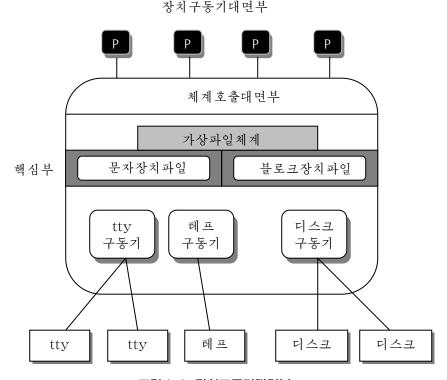



그림 1-4. 장치구동기대면부

어떤 사용자프로그람(p)이 하드웨어장치를 리용하려고 한다고 하자. 이 프로그람은 파일을 다루는 일반적인 체계호출과 보통 /dev 등록부에서 찾을수 있는 장치파일을 사용하여 핵심부에 요청을 할수 있다. 실제로 장치파일은 장치구동기대면부에서 사용자가 볼수 있는 부분이다. 각 장치파일은 특정한 장치구동기와 련결되여있어서 하드웨어구성요소에 작업을 요청하면 핵심부가 장치구동기를 호출하게 된다.

Unix가 등장했을 때에는 도형처리가 가능한 말단이 드물었으며 매우 비쌌기때문에 Unix핵심부는 자모와 수자를 지원하는 말단만 직접 다루었다. 그러나 도형처리가 가능한 말단이 널리 보급되여 X Windows체계와 같은 특별한 응용프로그람은 도형대면부카드의 입출력포구와 비데오기억령역에 직접 접근하면서도 표준프로쎄스로 동작하였다. Linux핵심부 2.4이상을 포함한 최신의 Unix핵심부중 일부는 도형카드의 프레임완충기(frame buffer)를 추상화하여 응용프로그람이 도형대면부카드의 입출력포구에 관해 아무것도 모르더라도 비데오카드에 접근할수 있게 한다.

제 3 절. 핵심부프로그람의 기초

Linux핵심부는 C와 아쎔블러어로 작성되여있다. 핵심부를 분석하는데서 이러한 언어의 특성들을 잘 아는것이 중요하므로 이 절에서는 여기에 대하여 실제적인 핵심부 2.6.9판의 원천코드를 기초로 론의를 진행한다.

1. Intel x86 CPU계렬의 주소지정방식

Linux핵심부는 각이한 기본방식에 대응하고있지만 가장 보편적인것이 Intel계렬이므로 이 기본방식의 중요요소들이 프로그람적으로 어떻게 실현되는가를 보기로 한다.

토막등록기

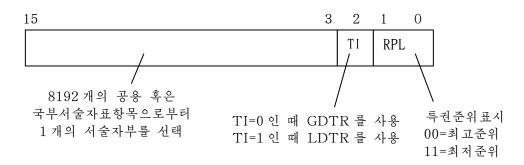



그림 1-5. 토막등록기의 자료마당

GDTR혹은 LDTR중의 토막서술자표지적자의 토막등록기의 값을 결합하여 구체적인 기억기안의 어느위치의 토막서술자인가를 결정한다.

```
typedef struct {
    unsigned short      seg_idx:13   /*13bit의 토막서술자항목지정*/
    unsigned short      ti:1;        /*토막서술자표를 가리키는 비트*/
    unsigned short      rpl:2;       /*요구특권준위*/
}토막등록기;
```

토막서술자표

<table>
<tr><td colspan="4">B31 – B24</td><td>G</td><td>D</td><td>A</td><td></td><td>L19 – L16</td></tr>
<tr><td>P</td><td>DPL</td><td>S</td><td>TYPE</td><td colspan="5">B23 – B16</td></tr>
<tr><td colspan="9">B15 – B0</td></tr>
<tr><td colspan="9">L15 – L0</td></tr>
</table>

B31–B24, B23–B16, B15–B0 : 기준주소(32bit)
L19–L16, L15–L0 : 토막크기(20bit)

그림 1-6. 토막서술자표의 구성

```
typedef struct {
unsigned int base24_31:8;   /*기준주소의 제일 웃 8bit/
unsigned int g:1;           /*토막크기단위비트. 0-바이트, 1-4kB */
unsigned int d_b:1;         /*기정동작크기존재방식 0=16, 1=32단위*/
unsigned int avl:1;         /*유효, 체계쏘프트웨어사용이 가능*/
unsigned int unused:1;              /*고정설치는 0*/
unsigned int seg_limit_16_19:4;     /*토막크기*/
  unsigned int p:1;  /*토막이 존재, 0일때 해당토막의 내용이 기억기에 존재하
지 않음*/
unsigned int dpl:2;  /*서술자특권준위*/
  unsigned int s:1;      /*서술자항목의 류형, 1-체계, 0-코드 혹은 자료구조*/
   unsigned int type:4;             /*토막의 류형, 우의 s와 함께 사용*/
   unsigned int base_0_23:24;       /*기준주소의 아래 24bit*/
   unsigned int seg_limit_0_15:16;  /*토막크기의 아래 16bit*/
}토막서술자항목;
```

기준주소가 높은 8bit, 아래 24bit로 갈라진것은 기동시 Intel에서는 24bit주소공간이고 그후 32bit의 주소공간으로 되기때문이다.

g비트가 1일 때 크기단위는 4kB이며 토막크기바이트의 아래 16bit의 용량은 64K이므로 한 토막의 최대가능한 크기는 64K×4K=256M이며 이것이 24bit주소공간의 크기로 된다.

CPU는 새토막등록기내용 및 GDTR 혹은 LDTR내용에 근거하여 대응하는 토막서술자항목을 찾아서 CPU에 넣는다. 이 과정에 CPU는 p비트(《present》를 표시)를 검사하여야 하는데 이 비트가 0이면 해당서술자항목이 가리키는 토막내용이 기억기에 없다는것을 표시하며 이때 CPU는 1차례외(exception, 새치기와 류사하다.)을 발생한다. 해당 봉사프로그람은 자기원판교환구획으로부터 이 토막의 내용을 기억기에 읽어들인 후 다시 p비트를 1로 설정하고 기억기에서 잠시 사용하지 않는 토막을 자기원판에 쓰기하고 그 토막의 서술자항목에서 p비트를 0으로 설정한다.

특권명령 LGDT는 0급상태에서만 사용할수 있다. 일반적으로 프로그람의 실행급수는 그 코드토막의 국부서술자항목(즉 토막등록기 CS에 의해 지정되는 국부토막서술자항목)안의 dpl에 의해 결정된다.

dpl: 서술자특권준위

매개 토막서술자항목의 dpl은 모두 0급상태에서 핵심부에 의해 결정된다.

토막등록기 CS의 ti비트가 1일 때 전체적인 토막서술자표를 사용해야 한다는것을 표시하며 0일 때에는 국부서술자표를 사용하며 rpl은 요구하는 권한을 표시한다.

2. i386의 페지화기억기관리

선형주소

80386은 선형주소공간을 4kB의 페지로 분할하며 매 페지는 물리기억공간의 임의의 4kB크기의 구간을 반영한다. 토막내에서 편위주소는 련속이며 선형주소는 련속이 아니다.

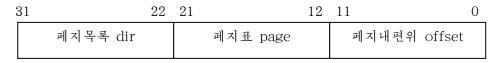



그림 1-7. 선형주소

```
typedef struct{
   unsigned int dir:10;       /*해당목록항목에서 1개의 페지로 지정*/
      unsigned int page:10;   /*구체적인 페지표안의 주소, 해당 표항목은 1개의 물리페지를 지정*/
   unsigned int offset:12;    /*4kB페지내의 편위주소*/
}선형주소;
```

페지목록에는 2^{10}=1024개의 목록항목이 존재하며 매개 목록항목은 1개의 페지표를 지정한다. 매개 페지표에는 또 1024개의 페지서술자항목이 있다.

선형주소로부터 물리주소에로의 변환과정

① CR3등록기로부터 페지목록의 기준주소를 얻는다.

② 선형주소의 dir값으로부터 목록내의 해당 페지표기준주소를 얻는다.

③ 선형주소의 page값으로부터 페지표의 해당 페지서술자항목을 찾는다.

④ 페지서술자항목의 페지기준주소와 선형주소안의 offset값을 더하여 물리주소를 얻는다.

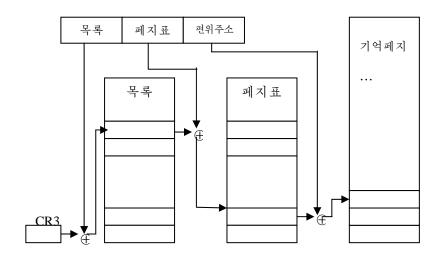



그림 1-8. 페지화과정

```
typedef struct{
unsigned int ptba:20;     /*페지표기준주소의 높은 20bit */
unsigned int avail:3;     /*체계프로그람작성자가 사용*/
unsigned int g:1;         /*global, 대역페지*/
unsigned int ps:1;        /*페지크기, 0은 4kB를 표시*/
unsigned int reserved:1;  /*예약, 영원히 0이다.*/
unsigned int a:1;         /*accessed, 이미 호출됨*/
unsigned int pcd:1;       /*닫긴(사용하지 않음)완충기억기*/
unsigned int pwt:1;       /*Write-Through, 완충기억기에 사용*/
unsigned int u_s:1;       /*0인 때 체계권한, 1인 때 리용자권한을 표시*/
unsigned int r_w:1;       /*읽기전용 혹은 쓰기가능*/
unsigned int p:1;         /*0인 때 해당페지가 기억기에 존재하지 않음*/
} 목록항;
```

3. 프로그람언어

1) C언어

Linux에서 핵심부의 기본부분은 GNU의 C언어로 씌여있으며 GNU가 제공하는 콤파일러는 gcc이다. gcc는 C++언어에서 《inline》과 《const》를 받아들였다. GNU의 C와 C++는 일체화되였으며 gcc는 C콤파일러인 동시에 C++콤파일러이다. 기능상 inline함수의 사용과 #define마크로정의는 서로 비슷한데 다만 상대적인 독립성을 가지며 또 더 안전하다. inline함수를 사용하면 프로그람검사에 유리하다.

64bit의 CPU구조를 지원하기 위해 gcc는 《long long int》라고 하는 일종의 새로운 기본자료류형을 확장하였다. gcc는 예약어를 가지지만 사용이 약간 다르다. 대부분의 C언어와 같이 gcc는 《aligned》, 《packed》등과 같은 속성서술기호를 제공한다. 이러한 서술기호는 C언어에서 새로운 예약어를 추가하는데 사용한다. 그러나 원래의 C언어에서(ANSI C와 같은) 이것은 비예약어를 발생하며 이러한것은 충돌을 발생시킬수 있다. 이를테면 gcc는 예약어 inline을 제공하며 《inline》은 원래 예약어가 아니므로(C++에서 예약어이다.) 오랜 코드에서는 이미 inline이라고 하는 변수이름을 가질수 있는데 이것은 충돌을 발생한다. 이러한 문제를 해결하기 위해서 gcc는 예약어로 사용하는 《inline》앞과 뒤에 《__》을 삽입하여 《__inline__》등으로 한다. 마찬가지로 《asm》에 대해서는 《__asm__》으로 한다.

gcc는 속성서술에 쓰이는 예약어 《attribute》를 가진다.

```
struct foo{
```

```
    char   a;
    int    x[z]   __attribute__ ((packed));
}
```

여기서 속성기호 《packed》는 문자 a와 옹근수배렬 x사이에서 공백으로 채우거나 남기는데서 32bit긴옹근수배렬한계에 따르지 않게 된다는것을 표시한다. 이렇게 《packed》는 바로 변수이름에서 충돌을 발생할수 있다. Linux핵심부의 판본은 gcc판본에 의존한다.

다음의 실례를 고찰하자.

```
#define DUMP_WRITE (addr,nr) do { memcpy(bufp,addr,nr), bufp += nr; } while(0)
```

do-while순환을 먼저 실행한 후 순환조건을 판단한다. 그러면 왜 이런 형태를 리용하는지 아래의 식을 살펴보기로 하자.

```
#define DUMP_WRITE (addr,nr) memcpy(bufp,addr,nr);\
    bufp += nr
```

다음

```
if(addr)
DUMP_WRITE (addr,nr);
else
   do_something_else();
```

은 다음과 같이 된다.

```
if(addr)
memcpy(bufp,addr,nr), bufp += nr;
else
   do_something_else();
```

이 경우 이 코드를 콤파일할 때 gcc는 실패할수 있으며 따라서 문법오유를 통지한다. 왜냐면 gcc는 memcpy()이후에 if문을 끝내기때문에 else로 넘어간다.

이를 위해

```
#define DUMP_WRITE (addr,nr) { memcpy(bufp,addr,nr);\
    bufp += nr; }

따라서
if(addr)
{ memcpy(bufp,addr,nr); bufp += nr; };
else
   do_something_else();
```

여기서는 《;》문에서 결속이 되므로 else는 오유로 된다. 따라서 do-while의 정의를 임의의 정황하에서 정의하여 문제를 없앤다.

2) 아쎔블러어

UNIX System V원천코드에서 3만행의 핵심부원천코드중에서 아쎔블러어는 약 2000행이 씌여져있다. 거의 20개의 .S와 .m의 확장자를 가진 파일로 갈라가며 그 중 대부분은 새치기와 이상처리를 위한 낮은 준위의 프로그람이다.

아쎔블러어코드를 리용하는 리유는 다음과 같다.

조작체계핵심부내의 낮은층 프로그람은 직접 하드웨어와 통신하며 일련의 전송지령을 리용하는데 C에는 이러한 지령이 없다. 실례로 386체계구조에서 inb, outb등과 같은 외부입출력지령에 대해 대응하는 C언어문구는 없다. 따라서 이러한 낮은층의 조작은 아쎔블러어를 써야 한다.

CPU의 일부 특수지령 즉 등록기들에 대한 조작과 새치기중지와 새치기발생과 같은 지령도 마찬가지인데 대응하는 C언어성분은 없다. 이외에 같은 종류의 체계구조의 각이한 CPU소편중에서 새로 개발하여 나온 CPU소편들 즉 Pentium, Pentium II, Pentium MMX에서는 특별히 새로운 지령들이 추가되였는데 이 지령에 대한 사용도 역시 아쎔블러어를 리용한다.

핵심부는 보조프로그람과 함수와 같은 일부 조작을 진행하는데 실행시에 자주 호출되는 이러한 것들에 대해서는 아쎔블러어를 리용하여 효률을 높일수 있다. 아쎔블러어를 리용하여 쓴 프로그람은 산법과 자료구조가 서로 같은 조건하에서 고급언어를 리용하여 쓴것에 비해 그 효률이 보통 높다.

다음으로 아쎔블러어를 쓰면 프로그람규모를 작게 할수 있다. 일부 특수한 경우에 보조프로그람의 공간도 아주 중요하게 나타난다. 그러한 대표적인 실례는 조작체계인도프로그람(Boot프로그람)이다. 체계의 인도프로그람은 보통 자기원판상의 첫번째 분구에 들어있다. 이것은 아쎔블러어로 되여있다.

그러면 Linux핵심부코드에 아쎔블러어로 씌여진 프로그람형태는 어떤가.

Unix에서는 AT&T가 정의한 형식의 아쎔블러어를 받아들였다. GNU가 개발한 각

종 체계프로그람은 AT&T의 386아쎔블러어형식을 계승하였으며 Intel의 형식을 받아들이지 않았다. 그러나 이 두개의 아쎔블러어사이에는 큰 차이가 없다.

Intel형식에서는 대문자를 많이 쓰지만 AT&T형식에서는 모두 소문자를 사용한다.

AT&T형식에서는 등록기명이 앞에 《%》가 붙는데 Intel형식에서는 그렇지 않다. AT&T의 386아쎔블러어중에서 지령의 원천연산수와 목표연산수는 Intel의 386아쎔블러어에서와 서로 반대이다. Intel형식에서는 목표는 앞에 있으며 원천은 뒤에 있다. 그러나 AT&T형식에서는 원천은 앞에 있으며 목표는 뒤에 있다. 실례로 등록기 eax의 내용을 ebx에 넣을 때 Intel형식에서는 《MOV EBX, EAX》이지만 AT&T형식에서는 《mov %eax, %ebx》로 된다. 보는것처럼 Intel형식의 설계자가 생각한것은《%eax->%ebx》이다.

AT&T형식에서는 지령내의 연산수크기(넓이)는 연산명령이름의 제일 마지막자모(연산명령의 제일 마지막 글자)로부터 결정된다. 연산명령뒤에 놓는 자모로는 b(8bit), w(16bit), l(32bit)을 가진다. Intel형식에서는 기억단위를 나타내는 연산수의 앞에는 《BYTE PTR》, 《WORD PTR》, 혹은 《DWORD PTR》를 붙여서 표시한다.

실례로

```
MOV     AL, BYTE PTR FOO (Intel형식)
movb    FOO, %al            (AT&T형식)
```

AT&T형식에서는 직접연산수가 앞글자로 《$》을 추가하여야 하는데 Intel형식에서는 앞글자에 아무것도 없다.

```
Intel형식 PUSH      4
AT&T형식 pushl $4
```

AT&T형식에서는 절대이행 혹은 호출지령 jump/call의 연산후 (즉시이행 및 호출의 목표주소)는 앞글자로 《*》을 추가한다. (대체로 C언어에서 지적자와 같이) 그러나 Intel형식에서는 없다.

원격이행지령과 원격보조프로그람호출지령의 연산명령의 이름은 AT&T형식에서는 《ljmp》혹은 《lcall》이며 Intel에서는 《JMP FAR》와 《CALL FAR》이다.

```
CALL    FAR   SECTION:OFFSET (Intel형식)
JMP     FAR   SECTION:OFFSET (Intel형식)

lcall   $section, $offset    (AT&T형식)
ljmp    $section, $offset    (AT&T형식)
```

되돌이지령은

```
RET     FAR   STACK_ADJUST    (Intel형식)
lret    $stack_adjust         (AT&T형식)
```

간접주소의 일반형식은 차이가 있다.

SECTION:[BASE+INDEX*SCALE+DISP]　(Intel형식)

section:disp(base,index,scale)　　　(AT&T형식)

3) C원천코드에 삽입하는 386아쎔블러어

C언어의 프로그람에 1토막의 아쎔블러어프로그람을 삽입할 때 gcc가 제공하는 《asm》단어기능을 사용할수 있다.

실례: include/asm/io.h

```
static inline void slow_down_io(void) {
   __asm__ __volatile__(
          __SLOW_DOWN_IO
#ifdef REALLY_SLOW_IO
          __SLOW_DOWN_IO __SLOW_DOWN_IO __SLOW_DOWN_IO
#endif
          : : );
}
```

여기서 asm과 volatile앞뒤에 《__》글자가 붙었는데 이것은 C언어에 대한 gcc의 일종의 확장이다.

같은 하나의 asm단어에 여러 행의 아쎔블러어프로그람을 삽입할수 있다. 그리고 각이한 조건에서 __SLOW_DOWN_IO는 또한 각이한 정의를 가진다.

```
#ifdef SLOW_IO_BY_JUMPING
#define __SLOW_DOWN_IO "jmp 1f; 1: jmp 1f; 1:"
#else
#define __SLOW_DOWN_IO "outb %%al,$0x80;"
#endif
```

우의 첫번째 경우에 목표 1f는 앞방향가기를 표시하는데 (f는 forward를 표시) 첫번째기호가 1인 행을 찾는다. 이와 마찬가지로 1b는 뒤방향탐색을 표시한다.

보다 더 정확한 리해를 위해 include/asm/atomic.h에 있는 다음의 함수를 고찰하자.

```
static __inline__ void atomic_add(int i, atomic_t *v)
{
   __asm__ __volatile__(
          LOCK "addl %1,%0"
          :"=m" (v->counter)
```

```
            :"ir" (i), "m" (v->counter));
    }
```

C코드에 삽입하는 아쎔블러어코드의 형식은 다음과 같다.

지령부:출력부:입력부:파괴부

여기서 《:》는 앞에서 고찰한 《1:》과 다르다. 첫번째 부분은 아쎔블러어구문의 본체이며 그 형식과 아쎔블러어프로그람에서 사용하는것은 서로 같으며 다만 구별이 있다. 이 부분은 반드시 있어야 한다.

지령부에서 수자는 앞에 %을 붙혀서 %0, %1로 하는데 이것은 등록기를 사용해야 하는 연산수를 표시한다. 사용가능한 형태의 연산수의 총수는 구체적인 CPU의 일반등록기개수에 의해 결정된다.

《출력부》는 출력변수에 대해 규정하는데 목표연산수는 결합의 약속조건으로 된다. 이러한 매개조건을 《약속(constraint)》이라고 한다.

:" =m" (v->counter)

여기서 약속 《 =m 》는 해당 목표연산수(지령부중의 %0)가 1개의 기억단위v->counter이라는것을 표시한다.

《입력부》에서는 입력약속과 출력약속은 서로 비슷한데 《=》기호가 없다.

실례에서 입력부는 2개의 약속을 가진다. 첫번째 《ir》(i)는 지령중의 %1이 등록기중의 하나의 《직접연산수》(i는 immediate을 표시)라는것을 표시하며 또 해당 연산수가 C원천코드중의 변수명(이 안에서는 호출참조이다.)i로 된다. 두번째 약속 《m》(v->counter)은 입력약속에서 의미가 서로 같다. 1개의 입력약속이 등록기사용을 요구하면 바로 전처리시에 gcc는 1개의 등록기를 분배하여 필요한 지령을 자동삽입하여 변수의 값은 해당 등록기에 들어간다.

아래의 표에서는 기본적인 약속조건들을 보여준다.

표 1-1. 주요약속조건을 표시하는 자모

자 모	의 미
“m”, “v”, “o”	기억기단위를 표시
“r”	임의의 등록기를 표시
“q”	등록기 eax,ebx,ecx,edx중의 하나를 표시
“i”, “h”	직접연산수를 표시
“E”, “F”	류점수를 표시
“g”	《임의》를 표시
“a”, “b”, “c”,	등록기 eax,ebx,ecx,edx를 사용하는가를 갈라서 표시

“d”	
“S” , “D”	등록기 esi 혹은 edi인가를 갈라서 사용할것을 요구
“I”	상수 《0부터 31》을 표시

1개의 연산수가 앞의 어떤 약속중에서 사용을 요구하여 요구하는것이 같은 등록기이면 그것은 곧 그 약속에 대응한 연산수기호를 약속조건에서 해방한다.

이외에 만일 조작후의 앞에서의 일부 약속에서 요구한것과 같은 등록기를 사용할것을 요구한다면 그 약속에 대응한 연산수쓰임번호를 약속조건에 놓는다.

파괴부에서는 항상 《memory》를 약속조건으로 놓는데 조작완성후에 기억기의 내용이 이미 변하였다는것을 나타내며 만일 원래의 일부 등록기(해당 체계의 조작에서 아직 쓰이지 않고)의 내용이 자기기억기로부터 온것이라면 현재 일치하지 않을수 있다. 주의해야 할것은 출력부가 빌 때 즉 출력약속이 없을 때 만일 입력약속이 존재하면 반드시 기호 《:》를 보류한다.

우의 실례는 변수 I의 값을 v->counter에 더하는것이다. 관건자 LOCK는 addl지령을 실행할 때 체계의 모선을 잠그어 다른 CPU(체계에 1개 CPU가 아니라면)로 하여금 다치지 못하게 하는것이다. C코드로서는 “v->counter += i;” 로 된다.

include/asm/bitops.h파일의 내용을 고찰하자.

```
#ifdef CONFIG_SMP
#define LOCK_PREFIX "lock ; "
#else
#define LOCK_PREFIX ""
#endif
#define ADDR (*(volatile long *) addr)
static inline void __set_bit(int nr, volatile unsigned long * addr)
{
   __asm__(
          "btsl %1,%0"
          :"=m" (ADDR)
          :"Ir" (nr));
}
```

지령 btsl은 1개의 32bit연산수들의 어떤 1bit를 1로 설정한다. 변수 nr는 해당 위치를 표시한다.

다음 실례로서 include/asm/string.h을 고찰하자.

```
static inline void * __memcpy(void * to, const void * from, size_t n)
{
int d0, d1, d2;
__asm__ __volatile__(
   "rep ; movsl\n\t"
   "testb $2,%b4\n\t"
   "je 1f\n\t"
   "movsw\n"
   "1:\ttestb $1,%b4\n\t"
   "je 2f\n\t"
   "movsb\n"
   "2:"
   : "=&c" (d0), "=&D" (d1), "=&S" (d2)
   :"0" (n/4), "q" (n), "1" ((long) to), "2" ((long) from)
   : "memory");
return (to);
}
```

여기서 《\n》은 행바꾸기부호이며 《\t》는 TAB부호를 표시한다. 우의 코드에서 지령부를 풀어서 보면 아래와 같다.

```
rep; movsl
   testb  $2, %b4
   je 1f
   movsw
1: testb  $1, %b4
   je 2f
   movsb
2:
```

제 2 장. 파일체계

제 1 절. 가상파일체계

Linux는 다른 체계와 공존할수 있는 특성이 있는데 Windows나 다른 Unix체계 심지어 Amiga와 같은 조작체계의 고유한 파일형식의 디스크나 구획도 탑재할수 있다.

Linux는 Unix체계와 마찬가지로 가상파일체계라는 개념을 사용하여 여러가지 디스크류형을 지원한다.

가상파일체계리면에 있는 사상은 여러가지 류형의 파일체계를 지원하기 위한 광범한 정보를 핵심부에 포함하는것이다.

핵심부에는 Linux에서 사용할수 있는 매개의 실제 파일체계가 제공하는 조작을 지원하기 위한 마당이나 함수가 있다.

핵심부는 매개의 read, write 혹은 기타 다른 함수의 호출에 대해 파일이 들어있는 파일체계에 따라 Linux고유의 파일체계, NT파일체계 또는 다른 파일체계의 실제함수로 대신한다.

이 절에서는 Linux가상파일체계의 목적, 구조, 실현 등을 다룬다. 다섯가지 표준 Unix파일류형중에서 정규파일(regular file), 등록부, 기호련결, 이 3가지를 집중적으로 본다.

1. 가상파일체계의 역할

《가상파일체계(VFS: Virtual File System)》는 표준Unix파일체계가 제공하는 모든 체계호출을 처리하는 핵심부쏘프트웨어계층이다. VFS의 우점은 여러 종류의 파일체계에 대해 공통대면부를 제공한다는 점이다.

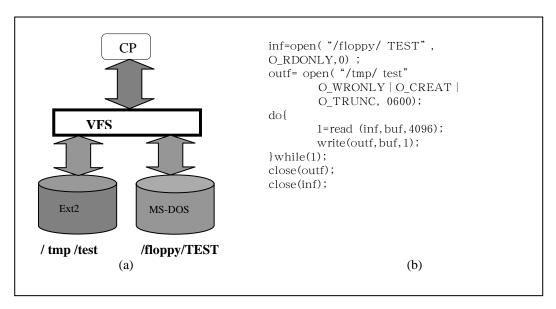



그림 2-1 단순한 파일복사작업을 위한 VFS 의 역할

례를 들어 어떤 사용자가 다음과 같은 쉘명령을 수행하였다고 하자.

```
$ cp /floppy/TEST /tmp/test
```

여기서 /floppy는 MS-DOS유연성자기원판의 탑재지적자이고 /tmp는 일반 Ext2(2차확장파일체계)등록부이다.

그림 2-1의 (a)에서 보는바와 같이 VFS는 응용프로그람과 파일체계실현사이의 추상계층이다.

따라서 cp프로그람은 /floppy/TEST와 /tmp/test의 파일체계류형을 알 필요가 없다.

대신 Unix프로그람작성을 해본 사람이라면 누구나 잘 알고있는 일반체계호출을 통해 VFS와 호상작용한다.

cp가 실행하는 코드는 그림 2-1에서 볼수 있는 (b)와 같다.

VFS가 지원하는 파일체계는 크게 다음과 같이 세 부류로 나눌수 있다.

1) VFS가 지원하는 파일체계

디스크기반의 파일체계

국부디스크구획의 기억장소를 관리한다. VFS가 지원하는 잘 알려진 디스크기반의 파일체계는 다음과 같다.

・널리 사용되는 Ext2, Ext3 그리고 ReiserFS와 같은 Linux의 독자적인 파일체계

・SYS V(System V, Coherent, XENIX)와 UFS(BSD, Solaris, NeXT), MINIX파일체계, VxFS(SCO Unix웨어) 등 다른 Unix를 위한 파일체계

・MS-DOS, FAT16, FAT32, NTFS(Windows95와 그 이후 배포판), NTFS(WindowsNT, Windows2000 , WindowsXP)같은 마이크로소프트파일체계

・ISO9660 CD-ROM파일체계, 통합디스크형식화(UDF, Universal Disk Format)DVD파일체계

・IBM OS/2(HPFS),Apple Macintosh (HFS), Amiga Fast파일체계(AFFS), ADFS 등 기타 전용파일체계

・IBM의 JFS, SGI의 XFS 등 Linux 이외의 체계에서 유래된 추가적인 기록형파일체계

망파일체계

망으로 련결된 다른 콤퓨터의 파일체계에 있는 파일에 쉽게 접근하게 해준다.

VFS가 지원하는 잘 알려진 망파일체계로는 NFS, Coda, AFS, SMB (Server Message Block, Microsoft Windows, IBM OS/2국부망 관리에서 파일과 인쇄기공유를 위해 사용한다), NCP(Novel Netware Core Protocol) 등이 있다.

특수파일체계

이 파일체계는 자신의 콤퓨터나 다른 콤퓨터의 실제 디스크공간을 관리하지 않는다.

/proc파일체계는 전형적인 특수파일체계이다.

Unix등록부는 뿌리가 /등록부인 나무구조를 만든다.

Linux에서 뿌리등록부는 대개 Ext2류형의 뿌리파일체계(root file system)에 위치한다.

다른 모든 파일체계는 뿌리파일체계의 보조등록부에 탑재된다.

디스크기반의 파일체계는 일반적으로 하드디스크, 유연성자기원판, CD-ROM 등의 하드웨어블로크장치에 저장한다.

Linux VFS가 제공하는 유용한 기능에는 /dev/loop0같은 《가상블로크장치(virtual block device)》를 다루는 기능이 있는데 정규파일에 저장된 파일체계를 탑재하는데 사용할수 있다.

활용가능한 응용분야의 례를 들면 사용자개인의 파일체계를 암호화해서 정규파일에 저장하여 자신의 개인적인 파일체계를 보호할수 있다.

최초의 가상파일체계는 1986년 Sun Microsystems의 SunOS에 포함되였다.

그 이후 Unix파일체계는 대부분 VFS를 포함한다.

그 가운데서도 Linux의 VFS는 가장 많은 파일체계를 지원한다.

2) 공통파일모형

VFS의 핵심개념은 지원하는 모든 파일체계를 표현할수 있는 《공통파일모형(common file model)》을 도입하는것이다.

이 모형은 전통적인 Unix파일체계가 제공하는 파일모형을 따른다.

그러나 매개의 특정 파일체계를 실현하려면 반드시 자신의 특정한 물리적인 구성을 VFS의 공통파일모형으로 변환해야 한다.

례를 들어 공통파일모형에서는 매 등록부를 파일의 목록과 다른 등록부들을 포함하는 파일처럼 생각한다.

그러나 일부 비Unix계렬의 디스크기반의 파일체계는 등록부나무구조에서 매 파일의 위치를 저장한 파일할당표(FAT)를 사용한다.

이런 파일체계에서 등록부는 파일이 아니다.

VFS의 공통파일 모형을 따르기 위해 FAT파일체계의 Linux실현에서는 필요하다면 실행중에 등록부에 대응하는 파일을 생성할수 있어야 한다.

이 파일은 핵심부 기억기객체로만 존재한다.

본래 Linux핵심부는 read()나 ioctl()연산을 처리하는 특정함수를 직접 실현(hardcode)할수 없다.

대신 각 연산에 대해 지적자를 사용해야 한다.

지적자는 접근할 특정파일체계를 위한 적절한 함수를 가리키게 한다.

이 개념을 설명하기 위해서 핵심부가 그림 2-1에서 본 read()를 MS-DOS파일체계에 해당하는 호출로 변환하는 방법을 살펴보자.

응용프로그람이 read()를 호출하면 핵심부가 sys_read()를 호출한다.

이 과정은 다른 체계호출에서도 마찬가지이다.

이 절의 뒤부분에서 볼수 있는것처럼 파일은 핵심부기억기에서 file자료구조체(linux/include/linux/fs.h에 정의)로 표현한다.

```
struct file {
    struct list_head      f_list;
    struct dentry                *f_dentry;
    struct vfsmount             *f_vfsmnt;
    struct file_operations       *f_op;/*해당 파일조작에 대한 지적자*/
    atomic_t              f_count;
    unsigned int                 f_flags;
    mode_t                       f_mode;
    int                   f_error;
    loff_t                f_pos;
    struct fown_struct f_owner;
    unsigned int          f_uid, f_gid;
    struct file_ra_state f_ra;
    unsigned long                f_version;
    void                  *f_security;
    /* 콘솔장치나 다른 장치를 위하여 리용*/
    void                  *private_data;

#ifdef CONFIG_EPOLL
    /* 이 파일에 대한 모든 후크를 리용하기 위하여
              fs/eventpoll.c에서 리용*/
    struct list_head      f_ep_links;
    spinlock_t            f_ep_lock;
#endif /* #ifdef CONFIG_EPOLL */
    struct address_space         *f_mapping;
};
```

이 자료구조에는 f_op라는 마당이 있어서 MS-DOS파일을 처리하는 함수에 대한 지적자를 포함한다.

이 함수가운데는 파일을 읽는 함수도 있다.

(linux/fs/fat/file.c에 선언)

```
struct file_operations fat_file_operations = {
    .llseek         = generic_file_llseek,
    .read           = generic_file_read, //읽기함수
    .write          = fat_file_write,
    .mmap           = generic_file_mmap,
    .fsync          = file_fsync,
    .readv          = generic_file_readv,
    .writev         = generic_file_writev,
    .sendfile       = generic_file_sendfile,
};
```

sys_read()는 이 함수에 대한 지적자를 찾아서 호출한다.
따라서 응용프로그람의 read()함수는 어느 정도 간접적인 호출로 바뀐다.

```
file->f_op->read(…);
```

이와 비슷하게 write()연산은 출력파일에 대응하는 적당한 Ext2쓰기함수를 실행하도록 한다. 요약하면 핵심부는 매개의 열린 파일에 대해 file변수에 알맞는 함수지적자를 할당하고 f_op마당이 가리키는 각 파일체계별로 호출을 실행해야 한다.

공통파일모형은 객체지향으로 볼수도 있는데 여기서 객체(object)는 자료구조와 이에 대한 연산을 수행하는 메쏘드를 한번에 정의하는 쏘프트웨어구조이다. 성능을 높이기 위해 Linux는 C++와 같은 객체지향 언어로 작성되지 않았다. 따라서 객체의 메쏘드에 대응하는 함수를 가리키는 항목을 포함한 자료구조로 객체를 실현한다.

공통파일모형은 다음과 같은 객체류형으로 이루어진다.

- **초블로크객체**

  탑재된 파일체계에 대한 정보를 저장한다. 디스크기반의 파일체계의 경우 이 객체는 일반적으로 디스크에 저장한 파일체계조종블로크 (file system control block)에 대응한다.

- **i마디객체**

  특정파일에 대한 일반정보를 저장한다. 디스크기반의 파일체계의 경우, 이 객체는 일반적으로 디스크에 저장한 파일조종블로크(file control block)에 대응한다.

  매 i마디객체에는 i마디번호가 할당되여 파일체계내에 있는 매 파일을 유일하게 식별한다.

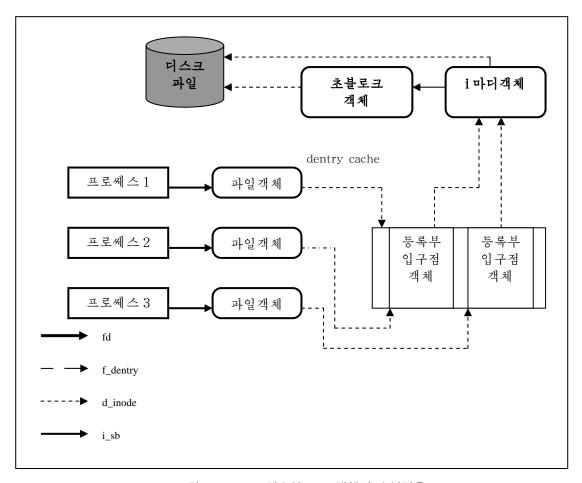



그림 2-2. 프로쎄스와 VFS 객체의 호상작용

파일객체

열린 파일과 프로쎄스사이의 호상작용과 관련한 정보를 저장한다. 이 정보는 각 프로쎄스가 파일에 접근하는동안 핵심부기억기에만 존재한다.

등록부입구점객체

등록부항목과 이에 대응하는 파일의 련결에 대한 정보를 저장한다.

매 디스크기반의 파일체계에서는 독자적인 방법으로 이 정보를 디스크에 저장한다.

그림 2-2는 프로쎄스가 파일과 호상작용하는 방법을 보여주는 간단한 례이다.

프로쎄스 3개가 파일하나를 연다. 그중 두 프로쎄스는 동일한 하드련결을 사용한다.

이 경우 세 프로쎄스는 자기의 파일객체를 소유한다.

그러나 등록부입구점객체는 매 하드련결당 하나만 필요하기때문에 두개가 존재

한다.

두 등록부입구점객체는 i마디객체 하나를 가리키고 i마디객체는 초블로크객체를 가리키며 함께 공통디스크파일을 구성한다.

VFS는 모든 파일체계에 대한 공통대면부를 제공하는일 외에도 체계성능과 관련한 중요한 역할을 한다.

가장 최근에 사용한 등록부입구점객체를 등록부입구점캐쉬(dentry cache)라는 디스크캐쉬에 저장한다.

이렇게 함으로써 파일경로명을 경로명의 마지막구성요소인 파일의 i마디로 변환하는 속도를 높인다.

일반적으로 디스크캐쉬(disk cache)는 보통디스크에 저장하는 정보의 일부를 기억기(RAM)에 저장하여 이후 이 정보에 접근할 때 더는 느린 디스크에 접근하지 않고 빨리 처리하도록 하는 쏘프트웨어기구이다.

등록부입구점캐쉬외에도 Linux에는 완충기캐쉬, 페지캐쉬와 같은 디스크캐쉬가 있는데 이것들에 대해서는 뒤에서 설명한다.

3) VFS가 처리하는 체계호출

표 2-1은 파일체계, 정규파일, 등록부, 기호련결을 대상으로 하는 VFS체계호출이다.

VFS가 제공하는 ioperm(), ioctl(), pipe(), mknod() 등 다른 체계호출은 장치파일이나 관(pipe)을 대상으로 하므로 뒤에서 보기로 한다.

VFS가 제공하는 socket(), connect(), bind(), protocols() 등의 체계호출은 소케트를 대상으로 하여 망기능을 실현한다.

표 2-1의 체계호출에 대응하는 핵심부봉사루틴은 이 절과 3절에서 다룬다.

표 2-1. VFS가 처리하는 체계호출

체계호출이름	설명
mount() umount()	파일체계탑재/해제
sysfs()	파일체계정보 얻음
statfs() fstatfs() ustat()	파일체계통계를 얻음
chroot() pivot_root()	뿌리등록부 변경
chdir() fchdir() getcwd()	현재등록부 변경
mkdir() rmdir()	등록부생성과 삭제
getdents() readdir() link() unlink() rename()	등록부입구점처리
readlink() symlink()	쏘프트련결 처리
chown() fchown() lchown()	파일소유자 변경

chmod() fchmod() utime()	파일속성 변경
stat() fstat() lstat() access()	파일상태 읽음
open() close()creat() umask()	파일을 열고 닫음
dup() dup2() fcntl()	파일서술자 처리
select() poll()	비동기적입출력 알림
truncate() ftruncate()	파일크기 변경
lseek()_llseek()	파일지적자 변경
read()write()readv()writev()sendfile()readahead()	파일입출력 연산실행
pread() pwrite()	파일탐색과 접근
mmap() munmap()madvise()mincore()	파일기억기배치처리
fdatasync() fsync() sync() msync()	파일자료동기화
flock()	파일잠그기 처리

앞에서 언급한바와 같이 VFS는 응용프로그람과 특정파일체계사이의 계층이다.

그러나 어떤 경우에는 저수준처리를 호출하지 않고 VFS가 독자적으로 파일연산을 실행할수 있다.

례를 들어 프로쎄스가 열린 파일을 닫으려 할 때에는 디스크의 파일에 접근할 필요가 없이 VFS에서 대응하는 파일객체를 해제한다.

또한 lseek()체계호출은 열린 파일과 프로쎄스의 호상작용과 관련한 속성인 파일지적자를 변경하는데 VFS는 대응하는 파일객체를 변경하는 작업으로 충분하며 디스크파일을 접근할 필요가 없다.

따라서 특정파일체계의 처리부를 호출할 필요도 없다.

또 다른 의미로 보면 VFS는 필요할 때에만 특정파일체계에 의존하는 《일반적인》 파일체계로 볼수 있다.

2. VFS자료구조

매 VFS객체는 객체속성과 객체메쏘드의 표에 대한 지적자로 이루어진 적절한 자료구조에 저장된다.

핵심부는 동적으로 객체의 메쏘드를 변경할수 있으며 특정객체에 대해 특수한 동작을 설정할수도 있다. 아래에 VFS객체와 이것들사이의 관계를 자세히 설명한다.

1) 초블로크객체

초블로크객체는 super_block구조체류형으로 되여있으며 표 2-2에서 설명하는 마당을 포함한다.

표 2-2. 초블로크객체의 마당

형	마당	설명
struct list_head	s_list	초블로크목록의 지적자
kdev_t	s_dev	장치식별자
unsigned long	s_blocksize	블로크크기(바이트단위)
unsigned char	s_blocksize_bits	블로크크기(비트단위)
unsigned char	s_dirt	변경된(불결한) 기발
unsigned long long	s_maxbytes	파일의 최대크기
struct file_system_type*	s_type	파일체계류형
struct super_operations*	s_op	초블로크조작지적자
struct dquot_operations*	dq_op	디스크할당조작지적자
struct quotactl_ops	s_qcop	초블로크할당조작지적자
struct export_operations	s_export_op	체계호출조작지적자
unsigned long	s_flags	탑재기발
unsigned long	s_magic	파일체계식별부호
struct dentry*	s_root	탑재등록부의 등록부입구점 객체
struct rw_scmaphore	s_umount	탑재해제를 위한 신호기
sruct scmaphore	s_lock	초블로크잠그기
int	s_count	참조계수기
atomic_t	s_active	2차 참조계수기
struct list_head	s_dirty	변경된 i마디목록
struct list_head	s_locked_inodes	I/O에 관련된 i마디목록
struct list_head	s_files	초블로크에 할당된 파일객체 목록
struct block_device*	s_bdev	블로크장치구동기서술자의 지적자
struct list_head	s_instances	해당 파일체계류형의 초블로크객체 목록의 지적자
struct quota_mount_options	s_dquot	디스크할당선택항목
union	u	특정파일체계정보

모든 초블로크객체(탑재된 파일체계마다 하나씩)는 원형2중련결목록(circular double linked list)으로 련결되여있다.

super_blocks변수가 목록의 처음 요소를 나타내며 초블로크객체의 s_list마당에는 목록에 린접한 요소의 지적자가 있다.

sb_lock스핀잠그기는 다중처리기(multi-processor)체계에서 목록에 동시에 접근하는것을 막는다.

마지막 u공용체(union)마당은 특정파일체계에 속한 초블로크정보를 포함한다.

례를 들어 초블로크객체가 Ext2파일체계를 가리키는 경우 이 마당에 ext2_sb_info구조체를 저장한다.

이 구조체에는 디스크할당비트마스크 등 VFS공통파일모형과 관계없는 기타 정보를 저장한다.

일반적으로 u마당의 정보는 효률을 높이기 위해 기억기에 복제된다. 모든 디스크기반의 파일체계는 디스크블로크를 할당하고 해제하기 위해 할당비트배치표를 읽고 변경해야 한다.

VFS는 이 파일체계들이 디스크에 접근하지 않고 기억기에 있는 초블로크의 u공용체마당을 직접 사용하도록 한다.

그러나 이 접근방법은 새로운 문제를 일으킨다.

VFS초블로크가 디스크의 대응하는 초블로크와 더는 동기화되지(synchronized) 않을수도 있다.

이것을 해결하기 위해 s_dirt기발을 도입하여 초블로크가 불결한(dirty)지, 즉 디스크의 자료를 갱신해야 하는지를 표시한다.

동기화가 취약하면 사용자에게 체계를 지운 상태로 끌(Shutdown) 기회를 주지 않고 갑자기 전원이 꺼진 경우 파일체계가 파괴되는 문제가 발생한다.

Linux는 주기적으로 모든 불결한 초블로크를 디스크에 기록함으로써 이 문제를 최소화한다.

초블로크와 관련한 메쏘드를 초블로크연산(superblock operation)이라고 한다. super_operations구조체로 이것을 표현하며 구조체의 주소는 s_op마당에 저장되여있다.

매개의 특정파일체계는 자기의 초블로크연산을 정의할수 있다.

VFS가 초블로크연산중 하나를 호출해야 할 때 실례로 read_inode()를 호출할 때에는 다음과 같이 실행한다.

```
sb->s_op->read_inode(inode);
```

여기서 sb는 초블로크객체주소를 저장하고있고 s_op는 super_operations구조체객체주소를 가리킨다.

s_op의 read_inode마당은 적절한 함수의 주소를 포함하므로 즉시 호출된다.

개별적인 조작들은 다음과 같다.

alloc_inode(struct super_block *sb)

새로운 inode 를 할당한다. sb는 해당 파일체계의 초블로크객체를 가르킨다.

read_inode(inode)

디스크에서 정보를 읽어서 inode가 가리키는 i마디객체의 마당을 채운다.

i마디객체의 i_ino마당값은 디스크에서 읽을 특정파일체계 i마디를 나타낸다.

dirty_inode(inode)

i마디가 변경되였음을 표시할 때 호출된다. ReiserFS와 Ext3와 같은 파일체계에서 디스크의 파일체계기록을 변경하기 위해서 사용한다.

write_inode(inode, flag)

inode가 나타내는 i마디객체내용으로 파일체계i마디를 갱신한다.

i마디객체의 i_ino마당값은 갱신하려는 파일체계i마디를 나타낸다.

flag변수는 I/O연산이 동기적(synchronous)인가를 나타낸다.

put_inode(inode)

inode가 나타내는i마디객체를 해제한다.

다른 프로쎄스가 이 객체를 여전히 사용하고있을수도 있으므로 객체를 해제하는 작업이 반드시 기억기변환을 의미하지는 않는다.

drop_inode(inode)

inode가 나타내는 i마디객체를 해제한다.

다른 프로쎄스가 이 객체를 여전히 사용하는 중이면 이 조작은 실패한다.

delete_inode(inode)

파일, 디스크i마디, VFS i마디를 포함한 자료블로크를 삭제한다.

put_super(super)

변수로 넘겨준 주소의 초블로크객체를 해제한다.(대응하는 파일체계를 탑재해제한 경우이다.)

write_super(super)

지정한 객체내용으로 파일체계의 초블로크를 갱신한다.

write_super_lockfs(super)

파일체계에 대한 변경을 차단하고 변수로 넘겨준 값을 사용하여 초블로크를 변경한다.

기록형파일체계가 이 메쏘드를 실현해야 하며 론리볼륨관리자(LVM, Logical Volume Manager)구동프로그람이 호출해야 한다. 현재는 사용되지 않는다.

unlockfs(super)

우의 초블로크메쏘드 write_super_lockfs가 파일체계변경을 차단한것을 취소한다.

statfs(super, buf, bufsize)

파일체계의 통계정보를 buf완충기에 기록한다.

remount_fs(super, flags, data)

새로운 항목으로 파일체계를 다시 탑재한다.(탑재항목을 변경해야 할 때 사용한다.)

clear_inode(inode)

put_inode와 비슷하지만 i마디가 나타내는 파일의 자료를 포함한 모든 페지를 해제한다.

umount_begin(super)

지정한 i마디에 대해 탑재해제연산을 시작했으므로 탑재연산을 중단한다.(망파일체계에서만 사용한다.)

show_options(seq_file, vfsmount)

파일체계의 항목을 표시하기 위해 사용한다.

우의 조작들은 모든 파일체계류형에 대해서 정의되여있다. 그러나 특정파일체계에는 실제로 이들중 일부만 사용할수 있다.

실례로 smb 파일체계의 초블로크조작구조체는 다음과 같이 정의되여있다.(linux/fs/smbfs/inode.c 에 정의)

```
static struct super_operations smb_sops =
{
   .alloc_inode = smb_alloc_inode,
   .destroy_inode      = smb_destroy_inode,
   .drop_inode  = generic_delete_inode,
   .delete_inode= smb_delete_inode,
   .put_super   = smb_put_super,
   .statfs      = smb_statfs,
   .show_options      = smb_show_options,
   .remount_fs  = smb_remount,
};
```

나머지 조작들은 정의되지 않으며 일부 조작들은 일반(generic_X_X)조작으로 정의되여있다.

일반조작은 모든 파일체계에 대하여 공통적인 조작들로서 VFS안에 정의되여있다.

초블로크를 읽는 read_super 메쏘드를 정의하지 않았다는 사실에 주목하자. (아직 디스크에서 읽어오지도 않은 객체의 매쏘드를 핵심부가 어떻게 호출할수 있는가?

read_super메쏘드는 파일체계류형을 서술하는 객체에 정의되여있다.)

2) i마디객체

파일체계가 파일을 다루는데 필요한 모든 정보는 i마디라는 자료구조에 들어있다.(linux/include/linux/fs.h 에 정의)

파일명은 림시로 할당한 이름표 같은것으로서 언제나 변경할수 있다.

그러나 i마디는 각 파일에 유일하며 파일이 존재하는 동안 그대로 남아있다.

기억기에 있는 i마디객체는 inode구조체로 이루어지는데 표 2-3에서 이 구조체의 마당을 설명한다.

표 2-3. i마디객체마당

형	마 당	설 명
struct list_head	i_hash	하쉬목록지적자
struct list_head	i_list	i마디목록지적자
struct list_head	i_dentry	등록부입구점목록지적자
unsigned long	i_ino	i마디번호
nusigned int	i_count	사용계수기
kdev_t	i_dev	장치식별자
umode_t	i_mode	파일류형과 접근권한
nlink_t	i_nlink	하드련결수
uid_t	i_uid	소유자식별자
gid_t	i_gid	그룹식별자
kdev_t	i_rdev	실제장치식별자
xoff_t	i_size	파일길이(B단위)
time_t	i_atime	최종파일접근시간
time_t	i_mtime	최종파일기록시간
time_t	i_ctime	최종i마디변경시간
unsigned int	i_blkbits	블로크크기(bit단위)
unsigned long	i_blksize	블로크크기(B단위)
unsigned long	i_blocks	파일의 블로크수
unsigned long	i_version	판본번호, 사용후 자동으로 증가
struct semaphore	i_sem	i마디신호기
struct semaphore	i_zombie	i마디제거나 i마디의 이름을 바꿀 때 사용하는 2차 i마디
struct inode_operations*	i_op	i마디연산
struct file_operations*	i_fop	기본파일연산
struct super_block*	i_sb	초블로크객체의 지적자
wait_qucuc_head_t	i_wait	i마디 대기렬
struct file_lock*	i_flock	파일잠그기목록지적자
struct address_spacc*	i_mappingaddr	객체지적자

	ess_space	
struct address_spacc*	i_data	블로크장치파일에 대한 address_space객체
struct dquot*	i_dquot	i마디디스크할당
struct list_head	i_devices	블로크장치파일 i마디목록의 지적자
struct pipe_inode_info*	i_pipe	파일이 관일 때 사용
struct block_device*	i_bdev	블로크장치구동기지적자
struct char_devece*	i_cdev	문자장치구동기지적자
unsigned long	i_dnotify_mask	등록부알림사건의 비트마스크
struct dnotify_struct*	i_dnotify	등록부알림에 사용됨
unsigned long	i_state	i마디상태기발
unsigned int	i_flags	파일체계탑재기발
unsigned char	i_sock	파일이 소케트이면 참
atomic_t	i_writecount	쓰기중인 프로쎄스 사용계수기
unsigned int	i_attr_flags	파일생성기발
__u32	i_generation	i마디판본번호(일부 파일체계에서 사용)
union	u	특정 파일체계 정보

마지막 u공용체마당은 특정파일체계에 속하는 i마디정보를 저장하기 위해 사용한다.

례를 들어 i마디객체가 Ext2파일을 가리키는 경우 이 마당은 ext2_inode_info구조체를 저장한다.

매 i마디객체는 디스크i마디안에 포함된 일부 자료(례를 들면 파일에 할당된 블로크개수)를 중복하여 포함하고있다.

i_state마당값이 I_DIRTY_SYNC, I_DIRTY_DATASYNC 또는 I_DIRTY_PAGES와 같은 경우 해당 i마디는 불결한것으로 된다.

따라서 대응하는 디스크i마디를 반드시 갱신해야 한다.

I_DIRTY마크로를 사용하여 이 세 경우의 여부를 한번에 검사할수 있다.

i_state마당은 I_LOCK(i마디가 I/O전송에 사용되고있음)과 I_FREEING(i마디객체해제중), I_CLEAR(i마디객체가 더는 의미없음) 등 다른 값일수도 있다.

매 i마디객체는 다음과 같은 원형2중련결목록중 하나에 언제나 포함된다.

- 유효하며 사용하고있지 않은 i마디목록: 유효한 디스크i마디의 내용을 담고있지만 현재 이 i마디를 사용하고있는 프로쎄스는 없다. 이 i마디는 불결한것이 아니며 i_count마당의 값은 0이다. 이 목록의 처음과 마지막요소는 inode_unused변수의 next, prev마당을 참조한다.
  이 목록은 디스크캐쉬로 동작한다.
- 사용중인 i마디목록: 유효한 디스크i마디의 내용을 담고있으며 현재 어떤 프로쎄스가 사용하고있다.
  i마디는 불결한것이 아니며 i_count마당의 값은 정수이다. 목록의 처음과 마지막요소는 inode_in_use변수로 참조한다.
- 불결한 i마디목록: 목록의 처음과 마지막요소는 대응하는 초블로크객체의 s_dirty마당으로 참조한다.

우의 목록항목은 매 i마디객체의 i_list마당으로 서로 련결된다.

i마디객체는 inode_hashtable이라는 하쉬표에도 포함된다.

하쉬표는 핵심부가 i마디번호와 해당 파일을 포함한 파일체계에 대응하는 초블로크객체주소를 알고있을 때 i마디객체를 찾는 검색속도를 빠르게 해준다.

하쉬법은 충돌을 일으킬수 있으므로 i마디객체는 같은 위치로 하쉬하는 다른 i마디를 가리키는 정방향지적자를 가지는 i_hash마당을 포함한다.

이 마당은 같은 위치로 하쉬되는 i마디의 2중련결목록을 구성한다. 하쉬표는 초블로크에 할당되지 않는 i마디(소케트가 사용하는 i마디와 같은 경우)의 특별한 련결목록도 포함한다.

이 목록의 처음과 마지막요소는 anon_hash_chain변수로 참조한다.

i마디객체와 관련한 메쏘드를 《i마디연산(inode operation)》이라고 한다.

i마디연산은 inode_operations구조체로 표현하며 i_op마당에 주소를 넣는다.

(linux/incude/linux/fs.h)

다음은 inode_operations표에 렬거된 inode 조작을 서술한것이다.

create(dir, dentry, mode)

새로운 정규파일을 위한 디스크i마디를 생성하여 주어진 등록부에서 등록부입구점객체와 련결한다.

lookup(dir, dentry)

등록부입구점객체의 파일명에 대응하는 i마디를 등록부내에서 탐색한다.

link(old_dentry, dir, new_dentry)

dir등록부내에서 old_dentry가 나타내는 파일을 가리키는 새로운 하드련결을 생성한다.

새로운 하드련결은 new_dentry에 지정한 파일명을 사용한다.

unlink(dir, dentry)

등록부입구점객체가 지정한 파일의 하드련결을 등록부에서 삭제한다.

symlink(dir, dentry, symname)

새로운 기호련결을 위한 i마디를 생성하여 주어진 등록부에서 등록부입구점객체와 련결한다.

mkdir(dir, dentry, mode)

새로운 등록부를 위한 i마디를 생성하여 주어진 등록부에서 등록부입구점객체와 련결한다.

rmdir(dir, dentry)

등록부입구점객체의 파일명과 같은 보조등록부를 주어진 등록부에서 제거한다.

mknod(dir, dentry, mode, rdev)

새로운 특수파일을 위한 i마디를 생성하여 주어진 등록부에서 등록부입구점객체와 련결한다.

mode와 rdev변수에 각각 파일류형과 장치의 주번호(major number)를 지정한다.

rename(old_dir, old_dentry, new_dir, new_dentry)

old_entry가 나타내는 파일을 old_dir등록부에서 new_dir등록부로 이동한다.

새 파일명은 new_dentry가 가리키는 등록부입구점객체에 들어있다.

readlink(dentry, buffer, buflen)

등록부입구점이 지정한 기호련결에 대응하는 파일경로명을 buffer가 가리키는 기억기령역에 복사한다.

follow_link(inode, dir)

i마디객체가 지정하는 기호련결을 변환한다. 기호련결이 상대경로명이면 지정된 등록부에서 탐색연산을 시작한다.

truncate(inode)

i마디가 지정하는 파일의 크기를 변경한다. 이 메쏘드를 호출하기 전에 새로운 파일크기값을 i마디객체의 i_size마당에 설정해야 한다.

permission(inode, mask)

i마디가 지정하는 파일에 지정한 접근방식이 허용되는지 검사한다.

revalidate(dentry)

등록부입구점객체가 지정하는 파일의 캐쉬되여있던 속성을 갱신한다. (일반적으로 망파일체계에서 호출한다.)

setattr(dentry, lattr)

i마디속성을 설정하고 변경사건을 발생시킨다.

getattr(dentry, iattr)

망파일체계에서 캐쉬된 i마디속성값을 새로 읽어야 함을 알리는데 사용한다.
우에서 설명한 메쏘드는 모든 i마디와 파일체계류형에서 정의된다.
그러나 특정i마디와 파일체계에서는 이중 일부만 실제로 사용할수 있다.
실현되지 않은 메쏘드에 대응하는 마당값은 NULL로 설정한다.

3) 파일객체

파일객체는 프로세스가 열린 파일과 어떻게 호상작용하는지 나타낸다.
이 객체는 파일을 열 때 생성되며 file구조체로 구성된다.
(linux/include/linux/fs.h 에 정의)
표 2-4에서는 file구조체마당을 설명한다.
파일객체는 디스크에 대응하는 영상이 없으며 따라서 file구조체에는 파일객체변경여부를 나타내는 불결한 마당이 없다.

표 2-4. 파일객체의 마당

형	마 당	설 명
struct list_head	f_list	일반적인 파일객체목록의 지시자
struct dentry*	f_dentry	파일에 관련된 등록부입구점객체
struct vfsmount*	f_vfsmnt	파일을 포함하는 탑재된 파일체계
struct file_operations*	f_op	파일연산표의 지시자
atomic_t	f_count	파일객체의 사용계수기
unsigned int	f_flags	파일을 열 때 지정된 기발
mode_t	f_mode	프로쎄스접근방식
loff_t	f_pos	현재파일편위(파일지시자)
unsigned long	f_reada	미리읽기기발
unsigned long	f_ramax	미리읽기페지의 최대수
unsigned long	f_raend	마지막미리읽기후 파일지시자
unsigned long	f_ralen	미리읽기바이트수
unsigned long	f_rawin	미리읽기페지수
struct fown_struct	f_owner	신호를 통한 비동기적입출력을 위한 자료
unsigned int	f_uid	사용자UID
unsigned int	f_gid	사용자GID
int	f_error	망쓰기연산을 위한 오유코드
unsigned long	f_version	판본번호, 매번 사용후 마다 자동증가

void*	rivate_datatty	장치구동프로그람용으로 필요
struct kiobuf*	f_iobuf	직접접근완충기를 위한 서술자
long	f_iobuf_lock	직접I/O전송을 위한 잠그기

파일객체에 저장하는 가장 중요한 정보는 《파일지적자(file pointer)》이다.

파일지적자는 파일에서의 현재 위치를 나타내며 이곳에서 다음 연산이 실행된다.

여러 프로쎄스가 같은 파일에 동시에 접근할수 있기때문에 파일지적자를 i마디객체에 저장할수는 없다.

매 파일객체는 언제나 다음 원형2중련결목록중 하나에 들어간다.

- 사용중이 아닌(unused) 파일객체목록: 이 목록은 파일객체의 기억기캐쉬로, 그리고 초사용자를 위해 예약된 공간으로 동작한다.

  이 공간을 사용하여 초사용자는 체계의 동적기억기를 전부 사용하였다 하더라도 파일을 열수 있다.

  객체가 사용중이 아니므로 f_count 마당값은 0이다.

  목록의 첫번째 요소는 허수아비(dummy)이며 free_list변수에 주소가 저장되여있다.

  핵심부는 목록이 요소를 최소한 NR_RESERVED_FILES객체의 값만큼 저장하고있도록 한다.

  이 값은 일반적으로 10이다.
- 사용중이며 초블로크에 할당되지 않은 파일객체의 목록: 목록내에 있는 매 요소의 f_count마당은 1로 설정된다.

  목록의 첫번째 항목은 허수아비이고 anon_list변수에 저장되여있다.
- 사용중이며 초블로크에 할당된 파일객체의 목록: 매 초블로크객체는 s_files마당에 파일객체목록의 첫번째 요소를 저장한다. 따라서 서로 다른 파일체계에 속한 파일의 파일객체는 서로 다른 목록에 저장된다. 목록내에 있는 매 요소의 f_count마당은 파일객체를 사용중인 프로쎄스수의 합+1로 설정된다.

파일객체가 어떤 순간에 어느 목록에 있든 목록의 이전 항목과 다음 항목을 가리키는 지적자는 파일객체의 f_list마당에 저장된다.

files_lock신호기는 다중처리기체계에서 목록에 동시에 접근하는것을 방지한다.

사용중이 아닌 파일객체목록의 크기는 files_stat변수의 nr_free_files 마당에 저장한다.

VFS는 새로운 파일객체를 할당할 때 get_empty_filp()함수를 호출한다.

이 함수는 《사용중이 아닌》목록이 NR_RESERVED_FILES값보다 많은 항목을 포함하고있는지 검사하여 더 많은 항목이 있다면 이중 하나를 새로 연 파일에 사용한다.

그렇지 않으면 일반적인 기억기할당으로 대체한다.

files_stat변수에는 nr_files마당(모든 목록에 포함되여있는 파일객체의 수)와 max_files마당(할당될수 있는 최대파일객체의 수 즉 체계에서 동시에 접근할수 있는 파일의 수)도 있다.

《공통파일모형》에서 설명한바와 같이 매 파일체계에는 파일읽기쓰기와 같은 작업을 위한 자신만의 파일연산(file operation)집합을 포함한다.

핵심부는 i마디를 디스크에서 기억기로 적재할 때 파일연산에 대한 지적자를 file_operations구조체에 저장한다.

이 구조체의 주소는 i마디 객체의 i_fop마당에 저장한다. 프로쎄스가 파일을 열 때 VFS는 i마디에 저장한 주소로 새로운 파일객체의 f_op마당값을 초기화하여 이후 파일연산에 대해 호출할 때 이 함수를 사용할수 있도록 한다.

필요하다면 VFS는 새로운 값을 f_op에 저장하여 파일연산집합을 변경할수도 있다.

다음으로 file_operations표에 렬거되여있는 파일조작들을 설명한다.

llseek(file, offset, origin)

파일지적자를 갱신한다.

read(file, buf, count, offset)

파일에서 *offset위치부터 count바이트만큼 읽는다. *offset값 (일반적으로 파일지적자에 대응)이 증가한다.

write(file, buf, count, offset)

count바이트만큼 파일의 *offset위치에 쓴다. *offset값(일반적으로 파일지적자에 대응)이 증가한다.

readdir(dir, dirent, filldir)

dirent에 다음 등록부항목을 반환한다. filedir변수는 등록부입구점내 마당을 추출하는 보조함수의 주소를 포함한다.

poll(file, poll_table)

파일에 어떤 활동이 있는지 검사하고 어떤 동작이 발생할 때까지 다시 잠든다.

ioctl(inode, file, cmd, arg)

하드웨어장치에 명령을 전송한다. 이 메쏘드는 장치파일에만 사용할수 있다.

mmap(file, vma)

파일에서 프로쎄스의 주소공간으로 기억기배치를 수행한다.

open(inode, file)

새로운 파일객체를 생성하여 파일을 열고 이것을 지정한 i마디객체에 련결한다.

flush(file)

열린파일에 대한 참조를 닫을 때 즉 파일객체의 f_count마당값이 감소할 때 호출한다.

이 메쏘드가 실제 수행하는 작업은 파일체계에 따라 다르다.

release(inode, file)

파일객체를 해제한다. 열린 파일에 대한 마지막참조를 닫을 때 즉 파일객체의 f_count마당값이 0이 될 때 호출한다.

fsync(file, dentry)

파일의 완충기억된 모든 자료를 디스크에 기록한다.

fasync(fd, file, on)

신호를 통해 비동기적으로 입출력을 알리는 일을 활성화/비활성화한다.

lock(file, cmd, file_lock)

파일에 잠그기를 적용한다.

readv(file, vector, count, offset)

파일에서 자료를 읽어서 vector가 나타내는 여러 완충기에 나누어 저장한다. 완충기의 수는 count에 저장한다.

writev(file, vector, count, offset)

vector가 나타내는 여러 완충기의 자료를 파일에 기록한다. 완충기의 수는 count에 지정한다.

sendpage(file, page, offset, size, pointer, fill)

지정한 파일에서 다른 파일로 자료를 전송한다. 이 메쏘드는 소케트에서 사용한다.

get_unmapped_area(file, addr, len, offset, flags)

파일배치를 위해 아직 사용되지 않은 주소범위를 얻는다.(흐레임완충기기억기배치에 사용된다.)

우의 메쏘드는 모든 파일류형에 정의되여있다.

그러나 특정파일류형에는 이것들중 일부만 실제 사용할수 있다.

실현하지 않은 메쏘드에 대응하는 마당값은 NULL이다.

4) 등록부입구점객체

《공통파일모형》에서 언급한바와 같이 VFS는 매 등록부를 파일내용으로서 등록부목록이 들어있는 일반 파일로 간주한다.

등록부입구점을 기억기로 읽어온 다음에 VFS는 이 등록부입구점을 dentry구조체의 등록부입구점객체로 변환한다.

표 2-5에서 이 구조체의 마당을 설명한다. 핵심부는 프로쎄스가 탐색하는 경로명의 모든 구성요소에 대해 등록부입구점객체를 만든다. 등록부입구점객체는 각 구성요소를 대응하는 i마디와 련결해준다. 례를 들어 /tmp/test경로명을 탐색할 때 핵심부는 뿌리등록부 /를 위해 첫번째 등록부입구점객체, 뿌리등록부내에 있는 tmp입구점을 위해 두

번째 등록부입구점객체, /tmp등록부내에 있는 test입구점을 위해 세번째 등록부입구점객체를 만든다.

표 2-5. 등록부입구점객체마당

형	마 당	설 명
atomic_t	d_count	등록부입구점객체 사용계수기
unsigned int	d_flags	등록부입구점기발
struct inode*	d_inode	파일이름과 관련한 i마디
struct dentry*	d_parent	부모등록부의 등록부입구점객체
struct list_head	d_hash	하쉬표입구점내 목록지시자
struct list_head	d_lru	사용되지 않는 목록지시자
struct list_head	d_child	부모등록부에 포함된 등록부입구점객체목록의 지시자
struct list_head	d_subdirs	등록부의 경우 보조등록부의 등록부입구점객체지시자
struct list_head	d_alias	관련i마디(별명)목록
int	d_mounted	등록부입구점이 파일체계를 위한 탑재위치일 때에만 1로 설정되는 기발
struct qstr	d_name	파일이름
unsigned long	d_time	d_revaliadte메쏘드에서 사용
struct dentry_operations*	d_op	등록부입구점메쏘드
struct super_block*	d_sb	파일의 초블로크객체
unsigned long	d_vfs_flags	등록부입구점캐쉬기발
void*.	d_fsdata	파일체계에 따른 자료
unsigned char*	d_iname	짧은 파일이름공간

디스크에는 등록부입구점객체에 대응하는 영상이 없다.

따라서 dentry구조체에는 객체가 변경되였다는 사실을 나타내는 마당이 없다.

dentry_cache라는 《스랩할당자캐쉬(slab allocator cache)》에 등록부입구점객체를 저장한다.

kmem_cache_alloc()와 kmem_cache_free()를 호출하여 등록부입구점객체를 생성, 제거한다.

각 등록부입구점객체는 다음 4개의 상태가운데서 하나에 해당한다.

- 해제(free)

이 등록부입구점객체는 의미있는 정보를 담고있지 않으며 VFS가 사용하고있지 않다.

대응하는 기억기령역은 스랩할당자(slab allocator)가 관리한다.

- 사용중이 아님(unused)

이 등록부입구점객체는 핵심부에서 사용하지 않고있다. 객체의 d_count사용계수기는 0이다.

그러나 d_inode마당은 아직 관련i마디를 가리키고있다. 등록부입구점객체는 유효한 정보를 포함하지만 만약 기억기를 회수할 필요가 있다면 내용을 삭제할것이다.

- 사용중(in use)

이 등록부입구점객체는 핵심부에서 사용하고있다. d_count사용계수기는 정수이고 d_inode마당은 관련i마디객체를 가리키고있다. 등록부입구점객체는 유효한 정보를 담고있으며 이것을 제거할수 없다.

- negative

이 등록부입구점에 대응하는 i마디가 더는 존재하지 않는다. 대응하는 디스크i마디가 삭제되였거나 존재하지 않는 파일의 경로명을 처리하면서 생성된 등록부입구점이기때문이다.

등록부입구점객체의 d_inode마당은 null로 설정된다. 그러나 등록부입구점객체는 계속 등록부입구점캐쉬에 남아 같은 파일경로명에 대해 탐색연산을 빠르게 처리할수 있도록 한다.

부수(negative)라는 용어와는 달리 어떤 부수와도 관계가 없다.

- 등록부입구점캐쉬

디스크에서 등록부입구점을 읽어서 이에 대응하는 등록부입구점객체를 생성하는 일은 상당한 시간이 걸리기때문에 사용을 끝낸 다음에도 다시 사용할것은 등록부입구점객체를 계속 기억기에 유지하는것이 도움이 된다.

례를 들어 사용자는 파일을 편집하고 파일을 콤파일한 다음 다시 편집하고 인쇄하거나 복사하고 복사한 파일을 편집한다.

이러한 경우 동일한 파일에 반복하여 접근하게 된다.

Linux는 등록부입구점을 처리하는 효률을 최대로 높이기 위해 등록부입구점캐쉬를 사용하는데 이것은 두가지 자료구조로 구성되여있다.

- 사용중, 사용중이 아님 또는 negative상태에 있는 등록부입구점 객체집합
- 주어진 등록부와 파일명으로부터 이에 대응하는 등록부입구점객체를 얻기 위한 하쉬표를 일반적으로 요청한 객체가 등록부입구점캐쉬에 들어있지 않은 경우, 하쉬함수는 null값을 반환한다.

등록부입구점캐쉬는 《i마디캐쉬(inode cache)》에 대한 조종기역할도 한다. 사용

중이 아닌 등록부입구점에 대응하는 핵심부기억기의 i마디인 경우, 등록부입구점캐쉬가 계속 사용하고있으므로 제거되지 않는다. 따라서 i마디객체는 읽기쓰기기억기(RAM)에 남아있고 대응하는 등록부입구점을 통해 빠르게 참조할수 있다.

《사용중이 아닌》 등록부입구점은 2중련결목록인 《가장 오래전에 사용한(LRU, Least Recently Used)》목록에 삽입시간순서대로 저장한다.

바꿔 말하면 가장 후에 해제한 등록부입구점객체가 목록의 맨앞에 들어가며 사용한지 가장 오래된 등록부입구점객체일수록 목록의 끝에 위치한다.

핵심부는 등록부입구점캐쉬의 크기를 줄여야 할 때 목록끝부분부터 요소를 제거하며 최근에 사용한 객체들은 유지한다. LRU목록의 처음과 마지막요소의 주소는 dentry_unused변수의 next와 prev마당에 각각 저장된다.

등록부객체의 d_lru마당은 목록내의 린접한 등록부입구점 의 지적자를 포함한다.

《사용중》인 등록부입구점객체는 대응하는 i마디객체의 i_dentry마당이 나타내는 2중련결목록에 삽입된다.(매 i마디는 여러 하드련결을 가질수 있으므로 목록이 필요하다.)

등록부입구점객체의 d_alias마당은 목록내의 린접한 항목을 가리킨다.

두 마당은 struct list_head형태이다. 대응하는 파일에 대한 마지막하드련결이 삭제되면 《사용중》인 등록부입구점객체는 《negative》상태가 된다.

이 경우 등록부입구점객체는 《사용중이 아닌》 등록부입구점객체의 LRU목록으로 이동한다.

핵심부가 등록부입구점캐쉬크기를 줄일 때 《negative》인 등록부입구점은 점차 LRU목록의 끝으로 이동하여 점차적으로 해제된다.

dentry_hashtable배렬을 사용하여 하쉬표를 실현한다.

각 항목은 하쉬함수를 거치면 같은 하쉬표값으로 하쉬되는 등록부입구점들의 목록을 가리키는 지적자이다.

배렬의 크기는 체계의 기억기용량에 따라 다르다. 등록부입구점객체의 d_hash마당은 하쉬값이 같은 목록의 린접항목에 대한 지적자를 포함한다.

하쉬함수는 등록부와 파일명의 등록부입구점객체주소 두개로부터 하쉬값을 생성한다.

dcache_lock스핀잠그기는 다중처리기체계에서 등록부입구점캐쉬자료구조에 대한 동시접근을 방지한다.

d_lookup()함수는 변수의 등록부입구점객체와 파일명에 대해 하쉬표를 탐색한다.

등록부입구점객체에 관련된 메쏘드를 등록부입구점연산(dentry operation)이라고 한다.

이것들은 dentry_operations구조체로 표현하며 주소는 d_op마당에 있다.

어떤 파일체계는 독자적인 등록부입구점메쏘드를 정의하기도 하지만 마당값은 대개 NULL이며 이 경우 VFS는 이것들을 기본함수로 대체한다.

등록부입구점조작의 구조체를 보여준다.

(linux/inlcude/linux/dcache.h에 정의)

다음은 dentry_operations표의 순서대로 메쏘드를 렬거한것이다.

d_revalidate(dentry, flag)

파일경로명을 변환하기 위해 등록부입구점객체를 사용하기 전에 등록부입구점객체가 아직도 유효한지 결정한다.

기본 VFS함수는 아무일도 하지 않지만 망파일체계의 경우에 자신의 함수를 지정할수도 있다.

d_hash(dentry, name)

하쉬값을 생성한다. 이 함수는 등록부입구점하쉬표에 사용되며 파일체계마다 독자적인 하쉬함수이다.

dentry변수는 구성요소를 포함하는 등록부를 나타낸다. name변수는 탐색하려고 하는 경로명구성요소, 하쉬함수가 생성하는 값을 포함하는 구조체를 가리킨다.

d_compare(dir, name1, name2)

두 파일명을 비교한다. name1은 반드시 dir가 가리키는 등록부에 속해야 한다.

기본 VFS함수는 단순문자렬비교함수이다. 그러나 매 파일체계는 자체의 방법으로 이 메쏘드를 실현할수 있다.

례를 들어 MS-DOS는 대소문자를 구별하지 않는다.

d_delete(dentry)

등록부입구점객체에 대한 마지막참조가 삭제되면(d_count가 0이 될 때) 호출한다.

기본VFS함수는 아무일도 하지 않는다.

d_release(dentry)

등록부입구점객체를 해제할 때(스랩할당자에 반납) 호출한다. 기본 VFS함수는 아무일도 하지 않는다.

d_iput(dentry, ino)

등록부입구점객체가 《negative》가 될 때 즉 i마디를 잃을 때 호출한다.

기본 VFS함수는 i마디객체를 해제하기 위해 iput()를 호출한다.

3. 프로쎄스관련파일

앞에서 매 프로쎄스는 자신의 현재 작업 등록부와 뿌리등록부를 가진다고 설명하였다.

이 두가지 정보는 프로쎄스와 파일체계의 호상관계를 표현하기 위해 핵심부가 관리해야 하는 정보의 례다.

이 호상관계를 표현하기 위해 fs_struct구조체를 사용한다.(표 2-6참고)

매 프로쎄스서술자에는 프로쎄스의 fs_struct구조체를 가리키는 fs마당이 있다.

표 2-6. fs_struct구조체마당

형	마 당	설 명
atomic_t	count	이 구조체를 공유하는 프로쎄스의 수
rwlock_t	lock	구조체의 마당을 위한 읽기/쓰기 스핀잠그기
int	umask	파일을 열 때 접근권한설정을 위한 비트마스크
struct dentry*	root	뿌리등록부의 등록부입구점
struct dentry*	pwd	현재작업등록부의 등록부입구점
struct dentry*	altroot	모방하는 뿌리등록부의 등록부입구점 (x86방식에서는 NULL)
struct vfsmount*	rootmnt	뿌리등록부에 탑재된 파일체계객체
struct vfsmount*	pwdmnt	현재 작업등록부에 탑재된 파일체계객체
struct vfsmount*	altrootmnt	모방하는 뿌리등록부에 탑재된 파일체계객체 (x86 방식에서는 NULL)

두번째 구조체의 주소는 프로쎄스서술자의 files마당에 있는데 이 구조체는 프로쎄스가 연 파일을 나타낸다.

files_struct구조체로 되여있으며 표 2-7에 마당을 렬거하였다.

표 2-7. files_struct구조체마당

형	마 당	설 명
atomic_t	count	이 구조체를 공유하는 프로쎄스수
rwkick_t	file_lock	구조체의 마당을 위한 읽기/쓰기스핀잠그기
int	max_fds	허용되는 파일객체수의 최대값
int	max_fdset	허용되는 파일서술자수의 최대값
int	next_fd	지금까지 할당된 파일서술자의 최대값+1
struct file**	fd	파일객체지적자배렬의 지시자
fd_set *	close_on_exec	exec()를 통해 닫을 파일서술자의 지시자
fd_set *	open_fds	열린파일서술자의 지시자

fd_set *	close_on_exec_init	exec()에 의해 닫을 파일서술자의 초기집합
fd_set *	open_fds_init	파일서술자의 초기집합
struct file**	fd_array	파일객체지시자의 초기배렬

fd마당은 파일객체지적자배렬을 가리킨다.

배렬의 크기는 max_fds 마당에 저장한다. 대개 fd는 files_struct구조체의 fd_array마당을 가리킨다.

초기에 fd_array는 파일객체지적자 32개를 담고있다. 프로쎄스가 파일을 32개이상 열면 핵심부는 더 큰 파일지적자배렬을 할당하고 fd마당에 그 주소를 저장한다.

그리고 max_fds마당값도 갱신한다.

fd배렬에 속한 모든 파일에 대해 해당 배렬색인값을 《파일서술자 (file descriptor)》라고 부른다.

대개 배렬의 첫번째 요소(색인값0)는 프로쎄스의 표준입력을 나타내고 두번째 요소는 표준출력, 세번째 요소는 표준오유를 나타낸다.

Unix프로쎄스는 파일서술자를 주파일식별자로 사용한다.

dup(), dup2(), fcnd()체계호출을 사용하여 두 파일서술자가 같은 열린파일을 나타내도록 할수 있다.

즉 배렬의 두 요소가 같은 파일객체를 가리킨다.

사용자가 표준오유를 표준출력으로 내보내기 위해 2>&1과 같은 쉘문법을 사용할 때마다 이 기능을 사용하는것이다.

한 프로쎄스는 파일서술자를 NR_OPEN(일반적으로 1048576)개이상 사용할수 없다.

핵심부는 프로쎄스서술자의 rlim[RLIMIT_NOFILE]구조체에 파일서술자수의 최대값에 대한 동적인 제한을 가지고있는데 이 값은 대개 1024이다.

프로쎄스가 뿌리권한을 가지고있으면 이 값을 늘일수 있다.

open_fds마당의 초기값은 open_fds_init마당의 주소를 담는다.

open_fds_init마당은 현재 열린파일의 파일서술자를 식별할수 있는 비트배치표이다.

max_fdset마당은 이 비트배치표의 비트수를 저장한다.

fd_set자료구조는 1024bit자료구조를 가지므로 일반적으로 비트배치표크기를 확장할 필요는 없다.

그러나 핵심부는 파일객체배렬의 경우와 마찬가지로 확장이 필요한 경우 비트배치표의 크기를 동적으로 확장한다.

핵심부는 파일객체를 사용하기 시작할 때 fget()함수를 호출한다.

이 함수는 fd파일서술자를 변수로 받는다. 그리고 대응하는 파일이 있으면 대응하는

파일객체의 주소인 current->files->fd[fd]를 반환하고 fd에 대응하는 파일이 없으면 NULL을 반환한다.

대응하는 파일이 있을 경우 fget()는 파일객체의 사용계수기인 f_count값을 1 증가시킨다.

핵심부는 핵심부실행경로에서 파일객체사용을 끝낼 때 fput()함수를 호출한다.

이 함수는 파일객체의 주소를 변수로 받으며 파일객체의 사용계수기인 f_count값을 1 감소시킨다.

이 마당값이 0이 되면 이 함수는 파일연산에서 release메쏘드가 정의되여있다면 이 메쏘드를 호출하고 이어서 이 파일객체를 가리키는 등록부입구점객체를 해제한다.

그리고 i마디객체의 i_writeaccess마당값을 감소시키고(파일을 쓰기방식으로 열었다면) 마지막으로 파일객체를 《사용중》에서 《사용중이 아님》목록으로 옮긴다.

4. 파일체계류형

Linux핵심부는 여러가지 파일체계를 지원한다.

여기서는 먼저 Linux핵심부의 내부에서 중요한 역할을 하는 몇가지 특수류형의 파일체계를 살펴본다.

다음으로 파일체계등록 즉 파일체계류형을 사용하기 전에 반드시 실행해야 하며 일반적으로 체계초기화단계에서 실행되는 기본연산을 살펴본다.

파일체계가 등록되여야만 핵심부가 파일체계의 독자적인 함수들을 사용할수 있으며 해당 류형의 파일체계를 체계의 등록부나무구조에 탑재할수 있다.

1) 특수파일체계

망 또는 디스크기반의 파일체계를 통해 사용자들이 핵심부외부에 저장된 정보를 다룰수 있지만 특수파일체계는 체계프로그람이나 관리자가 핵심부의 자료구조를 다루거나 조작체계의 특수기능을 실현할수 있도록 한다. 표 2-8은 Linux에서 사용하는 주요특수파일체계에 대한 탑재지점과 간단한 설명이다.

표 2-8. 주요특수파일체계

이름	탑재지점	설명
bdev	없음	블로크장치
binfmt_misc	임의	기타 실행파일형식화
devfs	/dev	가상장치파일
devpts	/dev/pts	가상말단지원(Unix98 표준)
pipefs	없음	관
proc	/proc	핵심부자료구조에 대한 범용접근위치
rootfs	없음	bootstrap단계를 위해 텅빈 뿌리등록부 제공

shm	없음	IPC공유기억기령역
sockfs	없음	소케트
tmpfs	임의	림시파일 (바꾸지 않으면 기억기에서 관리함)

어떤 특수파일체계에는 고정된 탑재지점이 없다.(표에서 《임의》로 표현)

이런 파일체계는 사용자가 임의의 위치에 탑재해서 사용할수 있다.

어떤 특수파일체계는 아예 탑재지점이 없다.(표에서 《없음》으로 표현)

이 경우는 사용자를 위한것이 아닌 핵심부가 이 특수파일체계를 사용함으로써 VFS 계층코드를 재사용하기 위한것이다.

례를 들어 pipefs 특수파일체계를 사용함으로써 관을 FIFO파일과 동일하게 처리할수 있다.

특수파일체계는 물리적인 블로크장치에 대응되지 않는다.

그러나 핵심부는 탑재된 특수파일체계마다 주번호 0과 임의의(매 특수파일체계마다 다른) 부번호를 가진 가상의 블로크장치를 할당한다.

get_unnamed_dev()함수는 새로운 가상블로크장치식별자를 할당하여 되돌려주고 put_unnamed_dev()함수는 이것을 해제한다.

unnamed_dev_in_use배렬은 어떤 부번호를 사용하고있는지 기록하는 256bit마스크를 포함하고있다. 가상블로크장치식별자라는 개념을 싫어하는 핵심부개발자들도 있지만 핵심부는 가상블로크장치를 사용하여 특수파일체계와 정규파일체계의 구분없이 한가지 방법으로 처리할수 있다.

핵심부가 어떻게 특수파일체계를 정의하고 초기화하는지 《뿌리파일체계 탑재하기》에서 실제적인 실례를 보게 된다.

2) 파일체계류형등록

사용자들은 자신의 체계를 위해 핵심부를 콤파일할 때 필요한 모든 파일체계를 Linux에 설정하는 경우가 많다. 그러나 실제로는 파일체계를 위한 코드를 핵심부영상에 포함할수도 있고 동적으로 모듈로 적재할수도 있다. VFS핵심부안에 코드가 현재 포함되여있는 모든 파일체계류형을 관리해야 한다.

VFS는 파일체계류형등록(filesystem type registration)을 통해 이 작업을 수행한다.

등록된 각 파일체계는 file_system_type객체로 표현하며 이 객체의 매 마당은 표 2-9와 같다.

표 2-9. file_system_type객체마당

류 형	마 당	설 명
const char *	name	파일체계이름
Int	fs_flags	파일체계류형기발

struct super_block* (*)()	read_super	초블로크를 읽기 위한 메쏘드
struct module*	owner	파일체계를 구현하는 모듈의 지시자
struct file_system_type*	next	다음목록요소를 가리키는 지시자
struct list_head	fs_supers	초블로크객체목록의 머리부

모든 파일체계류형객체는 단일련결목록에 삽입된다.

file_systems변수가 목록의 첫번째요소를 가리키며 구조체의 next마당이 목록의 다음 요소를 가리킨다.

file_system_lock 읽기/쓰기 스핀잠그기가 전체 목록에 대한 동시사용을 방지한다.

fs_supers마당은 탑재된 파일체계가운데서 이 파일체계류형인 초블로크객체들의 목록의 머리부(첫번째 허수아비요소)를 가리킨다.

목록의 이전과 다음요소를 가리키는 련결은 초블로크객체의 s_instances마당에 저장된다.

read_super마당은 디스크장치에서 초블로크를 읽어서 대응하는 초블로크객체에 복사하는 파일체계류형의 독자적인 함수를 가리킨다.

fs_flags마당은 표 2-10과 같은 여러기발을 담고있다.

표 2-10. 파일체계류형기발

이 름	설 명
PS_REQUIRES_DEV	이 류형의 파일체계는 반드시 물리적디스크장치에 위치해야 한다
PS_NO_DCAEHE	더는 사용되지 않음
PS_NO_PRELIM	더는 사용되지 않음
PS_SINGLE	이 파일체계형에 대해 초블로크객체 하나만 존재할수 있다
PS_NOMOUNT	파일체계가 탑재지점을 가지지 않음(《특수파일체계》참고)
PS_LRTTER	탑재해제이후에 등록부입구점캐쉬를 소거함(특수파일체계에서 사용)
PS_ODD_RENAME	변경(rename)연산은 이동(move)연산임(망 파일체계에서 사용)

체계초기화과정에서 콤파일할 때 지정한 모든 파일체계에 대해 register_filesystem()함수를 호출한다.

이 함수는 대응하는 file_system_type객체를 파일체계류형목록에 삽입한다.

파일체계를 실현하는 모듈을 적재한 경우에도 register_filesystem()함수를 호출한다.

이 경우 모듈을 해제하면 unregister_filesystem()함수호출을 통해 등록을 해소시킬수도 있다.

get_fs_type()함수는 파일체계가름을 변수로 받으며 등록된 파일체계 목록의 name마당을 탐색하여 일치하는 file_system_type객체가 존재하는 경우 이 객체의 지적자를 반환한다.

5. 파일체계탑재하기

매 파일체계는 자기의 뿌리등록부가 있다.

어떤 파일체계의 뿌리등록부가 체계전체등록부나무구조의 뿌리일 때 이것을 《뿌리파일체계》라고 한다.

다른 파일체계는 체계의 등록부나무구조에 탑재된다. 다른 파일체계를 탑재하는 위치에 있는 등록부를 탑재위치(mount point)라고 한다.

전통적인 Unix계렬핵심부에서 매 파일체계는 한번만 탑재할수 있다. 다음과 같은 명령으로 /dev/fd0유연성자기원판에 들어있는 Ext2파일체계를 /flp등록부에 탑재하였다고 하자.

```
mount  t ext2 /dev/fd0 /flp
```

umount명령으로 파일체계를 탑재해제하기 전까지는 /dev/fd0에 대한 탑재명령은 실패할것이다.

그러나 Linux는 다르다. 즉 한 파일체계를 여러번 탑재할수 있다. 례를 들어 앞의 명령 다음에 다음과 같은 명령을 실행하면 Linux에서는 성공한다.

```
mount  t ext2  o ro /dev/fd0 /flp-ro
```

결과적으로 유연성자기원판에 들어있는 Ext2파일체계는 /flp와 /flp-ro에 두번 탑재되였고 따라서 /flp와 /flp-ro두가지 경로를 통해 이 파일체계의 파일에 접근할수 있다.(이 례에서 /flp-ro를 통한 접근은 읽기전용이다.)

물론 어떤 파일체계가 n번 탑재되면 각 탑재를 통해 얻은 탑재지점 n개를 통해 해당 파일체계의 뿌리등록부에 접근할수 있다.

하지만 같은 파일체계에 여러 경로를 통해 접근할수 있더라도 이 파일체계는 단지 하나일뿐이다.

따라서 몇번 탑재되였더라도 이것들에 대해 초블로크객체는 단 하나만 존재한다.

탑재된 파일체계들은 계층을 구성한다.

첫번째 파일체계의 탑재위치가 두번째 파일체계의 등록부이고 두번째 파일체계의 탑

재위치가 다시 세번째 파일체계의 등록부일수 있다.

탑재지점 하나에 여러번 탑재하는것도 가능하다.

같은 탑재지점에 새로 탑재된 파일체계는 이전에 탑재된 파일체계를 덮어버린다.

이전의 탑재에서 파일과 등록부를 사용하던 프로쎄스는 계속 사용할수 있다.

제일 우에 탑재된 파일체계를 제거하면 그 아래 탑재되였던 파일체계가 나타난다.

각 탑재연산에 대해 핵심부는 탑재위치와 탑재기발, 탑재될 파일체계와 다른 파일체계와의 관계 등을 기억기에 저장해야 한다.

이런 정보는 탑재된 파일체계서술자(mount file system descriptor)라는 구조체에 저장된다.

매 서술자는 vfsmount형태의 자료구조이고 마당은 표 2-11과 같다.

표 2-11. vfsmount자료구조의 마당

형	마 당	설 명
struct list_head	mnt_hash	하쉬표목록의 지시자
struct vfsmount *	mnt_parent	이 파일체계가 탑재되는 부모파일체계의 지시자
struct dentry *	mnt_mountpoint	이 파일체계의 탑재등록부의 등록부입구점 지시자
struct dentry*	mnt_root	이 파일체계 뿌리등록부의 등록부입구점 지시자
struct super_block*	mnt_sb	이 파일체계의 초블로크객체의 지시자
struct list_head	mnt_mounts	서술자의 부모목록의 머리부(이 파일체계)
struct list_ head	mnt_child	서술자의 부모목록의 머리부(부모 파일체계)
atomic_t	mnt_count	사용계수기
int	mnt_flags	기발
char*	mnt_devname	장치파일이름
struct list_head	mnt_list	서술자의 대역목록의 지시자

vfsmount자료구조는 다음과 같은 2중련결원형목록여러개로 이루어진다.

- 탑재된 모든 파일체계를 담고있는 대역(global)2중련결원형목록 vfsmntlist변수가 나타내는 목록의 머리부는 허수아비요소이다. 서술자의 mnt_list마당은 목록의 린접항목을 가리키는 위치를 담는다.

- 부모파일체계의 vfsmount서술자의 주소와 탑재지점등록부의 등록부입구점객체의 주소를 통해 참조하는 하쉬표. 하쉬표는 mount_hashtable배렬에 저장되며 배렬의 크기는 체계의 자유호출기억기크기에 따라 다르다. 하쉬표의 각 요소는 하쉬값이 같은 모든 서술자를 저장한 원형2중련결목록의 머리부를 저장한다. 서술자의 mnt_hash마당은 목록의 린접요소를 가리키는 위치를 저장한다.
- 탑재된 각 파일체계마다 모든 탑재된 자식파일체계를 포함하는 원형2중련결목록. 매 목록의 머리부는 탑재된 파일체계서술자의 mnt_mounts마당에 저장된다. 서술자의 mnt_child마당은 목록의 린접요소를 가리키는 위치를 담는다.

mount_sem신호기는 탑재된 파일체계객체의 목록에 대한 동시접근을 방지한다.

서술자의 mnt_flags마당은 탑재된 파일체계에서 어떤 종류의 파일을 어떻게 처리할지 지정한다. 기발은 표 2-12에 있다.

표 2-12. 탑재된 파일체계기발

이 름	설 명
MNT_ NOSUID	탑재된 파일체계에서 setuid와 setgid기발을 금지한다
MNT_NODEV	탑재된 파일체계에서 장치파일에 대한 접근을 금지한다
MNT_NOEXEC	탑재된 파일체계에서 프로그람실행을 금지한다

다음 함수들이 탑재된 파일체계서술자를 처리한다.

alloc_vfsmnt()

기능: 탑재된 파일체계서술자를 할당하고 초기화한다.

free_vfsmnt()

기능 : mnt가 가리키는 탑재된 파일체계서술자를 해제한다.

lookup_mnt(parent, mountpoint)

기능 : 하쉬표에서 서술자를 탐색하여 그 주소를 반환한다.

1) 뿌리파일체계 탑재하기

뿌리파일체계를 탑재하는 일은 체계초기화과정의 주요부분의 하나이다. Linux핵심부는 뿌리파일체계를 하드디스크구획, 유연성자기원판, NFS에 의해 공유된 원격파일체계, 읽기쓰기기억기에 있는 가상블로크장치 등 다양한 장소에 저장할수 있도록 허용하므로 뿌리파일체계탑재는 아주 복잡한 일이다.

설명을 간단히 하기 위해 뿌리파일체계가 하드디스크의 구획에 저장되여있다고 가정하자.

핵심부는 체계기동과정에서 ROOT_DEV변수에서 뿌리파일체계가 들어있는 디스크의 주번호를 얻는다.

뿌리파일체계는 핵심부를 콤파일할 때 또는 적절한 항목을 초기기동적재프로그람에 전달하여 /dev등록부의 장치파일에서 지정할수 있다.

뿌리파일체계의 탑재기발은 root_mountflags변수에 들어있다. 사용자는 콤파일된 핵심부영상에 대해 rdev외부프로그람을 사용하거나 초기기동적재프로그람에 적당한 항목을 전달하여 이 기발을 지정한다.

뿌리파일체계를 탑재하는 일은 다음 목록과 같은 2단계작업으로 이루어진다.

① 초기탑재위치로 사용할 빈 등록부를 제공하는 rootfs특수파일체계를 탑재한다.

② 실제 뿌리파일체계를 빈 등록부에 탑재한다.

왜 핵심부는 실제파일체계를 탑재하기 전에 rootfs파일체계를 탑재하는가? rootfs파일체계덕분에 핵심부가 쉽게 뿌리파일체계를 변경할수 있다. 어떤 경우 핵심부는 여러 뿌리파일체계를 하나씩 차례로 탑재하고 탑재해제한다. 레를 들어 어떤 배포판의 초기기동유연성디스크는 최소한의 구동프로그람을 포함한 핵심부를 RAM에 적재하고 RAM디스크에 저장된 최소파일체계를 뿌리로 탑재한다.

그리고 이 초기뿌리파일체계에 있는 프로그람이 체계의 하드웨어를 탐색하고(레를 들면 하드디스크가 EIDE인지, SCSI인지) 필요한 모든 핵심부모듈을 적재한 다음 물리적블로크장치에서 뿌리파일체계를 다시 탑재한다.

첫번째 단계는 체계초기화과정에서 init_mount_tree()함수가 실행한다.

```
struct file_system_type root_fs_type;
root_fs_type.name = “roosfs”;
root_fs_type.read_super = rootfs_read_super;
root_fs_type.fs_flags = FS_NOMOUNT;
register_filesystem(&root_fs_type);
root_vfsmnt = do_kern_mount(“rootfs”, 0, “rootfs”, NULL);
```

root_fs_type변수는 rootfs특수파일체계의 서술자객체를 저장한다.

이 객체의 마당은 초기화된 다음 register_filesystem()함수에 전달된다.

do_kern_mount()함수는 특수파일체계를 탑재하고 새로 탑재된 파일체계객체의 주소를 반환한다.

이 주소는 init_mount_tree()를 통해 root_vfsmnt변수에 저장된다.

이제 root_vfsmnt는 탑재된 파일체계나무의 뿌리를 나타낸다.

do_kern_mount()는 다음과 같은 변수를 받는다.

fstype

탑재될 파일체계류형

flags

탑재기발(《일반파일체계 탑재하기》에 있는 표 2-13 참고)

name

파일체계를 저장하는 블로크장치의 장치파일명(또는 특수파일체계의 파일체계류형이름)

data

파일체계의 read_super메쏘드에 전달할 추가적인 자료의 지적자

do_kern_mount()는 다음과 같은 연산을 통해 실제 탑재연산을 수행한다.

① 현재프로쎄스가 탑재연산을 수행할 권한을 소유하고있는지 검사한다.(체계초기화는 뿌리가 소유한 프로쎄스에 의해 수행되므로 함수가 init_mount_tree()에 의해 호출되면 언제나 성공한다.)

② get_fs_type()를 호출하여 type변수의 파일체계가름을 파일체계류형목록에서 탐색한다. get_fs_type()는 대응하는 file_system_type서술자의 주소를 반환한다.

③ alloc_vfsmnt()를 호출해서 새로 탑재된 파일체계서술자를 할당하여 그 주소를 국부변수 mnt에 저장한다.

④ mnt->mnt_devname마당을 name변수의 내용으로 초기화한다.

⑤ 새로운 초블로크를 할당하고 초기화한다. do_kern_mount()는 이것을 어떻게 수행할지 결정하기 위해 file_system_type서술자의 기발을 검사한다.

ㄱ. FS_REQUIRES_DEV가 설정되여있으면 get_sb_bdev()를 호출한다.(《일반파일체계 탑재하기》 참고)

ㄴ. FS_SINGLE이 설정되여있으면 get_sb_single()를 호출한다.(《일반파일체계 탑재하기》 참고)

ㄷ. 그렇지 않으면 et_sb_nodev()를 호출한다.

⑥ file_system_type서술자의 FS_NOMOUNT기발이 설정되여있으면 초블로크객체의 MS_NOUSER기발을 설정한다.

⑦ mnt->mnt_sb마당을 새로운 초블로크객체의 주소로 초기화한다.

⑧ mnt->mnt_root와 mnt->mnt_mountpoint마당을 파일체계의 뿌리등록부에 대응하는 등록부객체의 주소로 초기화한다.

⑨ mnt->mnt_parent마당을 mnt의 값으로 초기화한다.(새로 탑재된 파일체계는 부모가 없다.)

⑩ 초블로크객체의 s_umount신호기를 해제한다.(이 신호기는 5단계에서 객체를 할당할 때 획득한것이다.)

⑪ 탑재된 파일체계객체의 주소 mnt를 반환한다.

init_mount_tree()가 rootfs특수파일체계를 탑재하기 위해 do_kern_mount()를

호출할 때 FS_REQUIRES_DEV기발이나 FS_SINGLE기발 둘다 설정되여있지 않으므로 do_kern_mount()는 초블로크객체를 할당하기 위해 get_sb_nodev()를 사용한다.

init_mount_tree()함수는 다음과 같은 연산을 수행한다.

① get_unnamed_dev()를 호출하여 새로운 가상블로크장치식별자를 할당한다.(앞에서 《특수파일체계》 참고) 파일체계류형객체, 탑재기발, 가상블로크장치식별자를 변수로 read_super()를 호출한다. 이 함수는 다음 연산을 수행한다.

ㄱ. 새로운 초블로크객체를 할당하여 그 주소를 국부변수 s에 넣는다.

ㄴ. s->s_dev마당을 블로크장치식별자로 초기화한다.

ㄷ. s->s_flags마당을 탑재기발로 초기화한다.(표 2-13참고)

ㄹ. sb_lock스핀잠그기를 획득한다.

ㅁ. s->s_type마당을 파일체계의 파일체계류형서술자로 초기화한다.

ㅂ. super_blocks가 머리부를 가리키는 파일체계류형목록에 초블로크를 삽입한다.

ㅅ. s->s_type->fs_supers가 머리부를 가리키는 파일체계류형목록에 초블로크를 삽입한다.

ㅇ. sb_lock스핀잠그기를 해제한다.

ㅈ. 쓰기를 위해 s->umount읽기/쓰기 신호기를 획득한다.

ㅊ. s->s_lock신호기를 획득한다.

ㅋ. 파일체계류형의 read_super메쏘드를 호출한다.

ㅌ. s->s_flags의 MS_ACTIVE기발을 설정한다.

ㅍ. s->s_lock신호기를 설정한다.

ㅎ. 초블로크의 주소 s를 반환한다.

② 핵심부모듈에 실현이 들어있는 파일체계류형이라면 사용계수기를 1 증가시킨다.

③ 새로운 초블로크의 주소를 반환한다.

뿌리파일체계의 탑재연산에서 두번째 단계는 체계초기화 끝부분에서 mount_root()함수가 처리한다.

설명을 간단히 하기 위해 디스크기반의 파일체계에서 장치파일을 전통적인 방법으로 처리하는 경우로 본다.

이 경우 함수는 다음연산을 수행한다.

① 완충기를 할당하여 파일체계류형이름목록으로 채운다. 이 목록은 rootfstype기동변수를 통해 핵심부에 전달되거나 파일체계형태의 단순련결목록을 탐색하여 만든것이다.

② bdget()나 blkdev_get()함수를 호출하여 뿌리장치 ROOT_DEV가 존재하고 잘

동작하는지 검사한다.

③ get_super()를 호출하여 super_blocks목록에서 ROOT_DEV장치에 대응하는 초블로크객체를 찾는다.

아직 뿌리파일체계가 탑재되지 않았으므로 아무것도 찾을수 없다.

그러나 이전에 탑재된 파일체계를 다시 탑재하는것이 가능하므로 검사를 한다.

뿌리파일체계는 체계기동과정에서 보통 두번 탑재된다. 첫번째는 파일체계의 완전성을 안전하게 검사하기 위해 읽기전용으로 탑재된다.

두번째는 정상적으로 디스크를 사용할수 있도록 읽기/쓰기용으로 탑재된다.

여기서는 super_blocks목록에서 ROOT_DEV장치에 대응하는 초블로크객체를 찾지 못하였다고 가정한다.

④ ①에서 만든 파일체계류형이름목록을 탐색한다.

각 이름에 대해 get_fs_type()를 호출하여 대응하는 file_system_type객체를 얻는다.

그리고 read_super()를 호출하여 대응하는 초블로크를 디스크에서 읽기를 시도한다.

앞에서 설명한것처럼 이 함수는 새로운 초블로크를 할당하고 file_system_type객체의 read_super마당이 가리키는 메쏘드를 사용하여 블로크를 채우려고 시도한다.

각 파일체계의 독자적인 메쏘드가 유일한 식별번호(magic number)를 가지기때문에 모든 read_super()호출은 뿌리장치가 실제 사용하는 파일체계의 메쏘드를 초블로크를 채우려고 하는 경우외에는 모두 실패한다.

read_super()메쏘드는 뿌리등록부에 대한 i마디객체와 등록부입구점객체도 생성한다.

등록부입구점객체는 i마디객체에 대응한다.

⑤ 새로 탑재된 파일체계객체를 할당하고 ROOT_DEV블로크장치이름, 초블로크객체의 주소, 뿌리등록부의 등록부입구점객체의 주소로 마당을 채운다.

⑥ graft_tree()함수를 호출하여 새로 탑재된 파일체계객체를 root_vfsmnt의 자식목록, 탑재된 파일체계객체의 대역목록, mount_hashtable하쉬표에 삽입한다.

⑦ 현재프로쎄스(init프로쎄스)의 fs_struct구조체의 root와 pwd 마당을 뿌리등록부의 등록부입구점객체로 초기화한다.

2) 일반파일체계 탑재하기

뿌리파일체계를 초기화하면 다른 파일체계를 탑재할수 있다. 매 파일체계는 자신의 탑재위치가 있어야 하는데 탑재위치는 체계의 등록부나무에 존재하는 등록부 하나면 충분하다.

파일체계를 탑재하기 위해 mount()체계호출을 사용한다.

이 체계호출의 sys_mount()봉사루틴은 다음과 같은 변수를 받는다.

- 파일체계가 들어있는 장치파일의 경로명. 이것이 필요하지 않다면 NULL(레를 들면 탑재될 파일체계가 망기반인 경우)
- 파일체계를 탑재할 등록부의 경로명(탑재지점)

- 파일체계류형. 반드시 등록된 파일체계의 이름이여야 한다.
- 탑재기발(사용가능한 값은 표 2-13에 있다.)
- 파일체계에 따른 자료구조지적자(NULL일수도 있다.)

표 2-13. 탑재기발

마 크 로	설 명
MS_RDONLY	파일을 읽을수만 있음
MS_NOSUID	setuidfi나setgid기발금지
MS_NODEV	장치파일접근금지
MS_NOEXEC	프로그람실행금지
MS_SYNCHRONOUS	쓰기연산을 즉시 실행
MS_REMOUNT	탑재기발을 변경하여 파일체계를 다시 탑재
MS_MANDLOCK	강제적(mandatory)잠그기허용
MS_NOATIME	파일접근시간을 갱신하지 않음
MS_NODIRATIME	등록부접근시간을 갱신하지 않음
MS_BIND	결합(bind)지점을 생성하여 파일이나 등록부가 체계등록부나무의 다른 위치에 보이게 함
MS_MOVE	탑재된 파일체계를 다른 탑재지점으로 이동
MS_REC	등록부보조나무에 대해 속박탑재를 재귀적으로 생성
MS_VERBOSE	탑재오유에 대해 핵심부통보문생성

sys_mount()함수는 변수의 값을 림시핵심부완충기에 복사하고 큰 핵심부잠그기를 획득한 다음 do_mount()함수를 호출한다.

do_mount()에서 돌아오면 봉사루틴이 큰 핵심부잠그기를 풀어주고(release) 림시핵심부완충기를 해제(free)한다.

do_mount()함수는 다음과 같은 연산을 통해 실제 탑재연산을 수행한다.

① 탑재기발의 웃 16bit가 《magic》값 0xce0d로 설정되여있는지를 검사한다. 이 경우 이 부분을 0으로 지운다. 이것은 sys_mount() 봉사루틴이 웃자리비트기발을 처리하지 못하는 오래된 C서고와 함께 동작할수 있도록 하는 오래된 기법이다.

② 변수에 MS_NOSUID, MS_NODEV또는 MS_NOEXEC 기발이 설정되여있으면 이것들을 지우고 탑재된 파일체계객체의 대응하는 기발(MNT_NOSUID, MNT_NODEV, MNT_NOEXEC)을 설정한다.

③ path_lookup()과 path_walk()를 호출하여 탑재지점의 경로명을 탐색한다.

④ 탑재기발을 검사하여 어떤 작업을 수행할것인가를 결정한다.

ㄱ. MS_REMOUNT기발이 설정되여있으면 초블로크객체의 s_flags마당의 탑재기발과 탑재된 파일체계객체의 mnt_flags마당의 탑재된 파일체계기발을 변경해야 한다. do_remount()함수가 이 변경을 수행한다.

ㄴ. 그렇지 않으면 MS_BIND기발을 검사한다. 기발이 설정되여있으면 사용자가 파일이나 등록부를 체계등록부나무의 또 다른 위치에 보이게 하려는것이다. 이것은 보통 물리적디스크구획이 아닌 일반파일에 저장된 파일체계를 탑재할 때 사용한다.(loopback) do_loopback()함수가 이 작업을 수행한다.

ㄷ. 그렇지 않으면 MS_MOVE기발을 검사한다. 기발이 설정되여있으면 사용자는 이미 탑재된 파일체계의 탑재위치를 변경을 요청한것이다. do_move_mount() 함수가 이것을 수행한다.

ㄹ. 그렇지 않으면 do_add_mount()를 호출한다. 이 경우가 가장 일반적이다. 이 경우 사용자가 디스크구획에 저장된 특수파일체계나 일반파일체계의 탑재를 요청한것이다. do_add_mount()는 다음과 같은 작업을 수행한다.

ㅁ. 파일체계류형, 탑재기발, 블로크장치이름을 변수로 do_kern_mount()를 호출한다.《뿌리파일체계 탑재하기》에서 설명한것처럼 do_kern_mount()가 실제탑재연산을 처리한다.

ㅂ. mount_sem신호기를 획득한다.

ㅅ. do_kern_mount()가 할당한 새로 탑재된 파일체계객체의 mnt_flags마당의 기발을 초기화한다.

ㅇ. graft_tree()를 호출하여 새로 탑재된 파일체계 객체를 대역목록, 하쉬표, 탑재된 부모파일체계의 자식목록에 삽입한다.

ㅈ. mount_sem신호기를 해제한다.

④ path_release()를 호출하여 탑재위치의 경로명탐색을 끝낸다. (뒤에 나오는 《경로명탐색》참고)

앞에 있는 《 뿌리파일체계 탑재하기 》에서 이미 본것처럼 탑재연산의 핵심은

do_kern_mount()함수이다.

이 함수가 파일체계류형기발을 검사하여 어떤 탑재연산을 실행할것인가를 결정한다는 사실을 기억할것이다.

일반디스크기반의 파일체계의 경우 FS_REQUIRES_DEV기발이 설정되며 따라서 do_kern_mount()는 get_sb_bdev()함수를 호출하여 다음 작업을 수행한다.

① path_lookup()과 path_walk()를 호출하여 블로크장치(block device)의 경로명을 탐색한다.(《경로명탐색》참고)

② blkdev_get()를 호출하여 일반파일체계를 저장하고있는 블로크장치를 연다.

③ 초블로크객체의 목록을 탐색한다. 블로크장치에 대한 초블로크가 이미 있으면 그 주소를 반환한다. 파일체계가 이미 탑재되여있으며 다시 탑재되는 경우이다.

④ 그렇지 않으면 새로운 초블로크객체를 할당하고 s_dev, s_bdev, s_flags, s_type마당을 초기화하고 객체를 초블로크의 대역목록, 파일체계류형서술자의 초블로크목록에 삽입한다.

⑤ 초블로크의 s_lock스핀잠그기를 획득한다.

⑥ 파일체계류형의 read_super메쏘드를 호출하여 디스크의 초블로크정보를 읽고 새로운 초블로크객체의 다른 마당들을 채운다.

⑦ 초블로크의 MS_ACTIVE기발을 설정한다.

⑧ 초블로크의 s_lock스핀잠그기를 해제한다.

⑨ 핵심부모듈에 실현된 파일체계류형인 경우 사용계수기를 1 증가시킨다.

⑩ 탑재위치탐색연산을 끝내기 위해 path_release()를 호출한다.

⑪ 새로운 초블로크객체의 주소를 반환한다.

3) 파일체계탑재해제하기

파일체계를 탑재해제하기 위해 umount()체계호출을 사용한다.

이 체계호출에 대응하는 sys_umount()봉사루틴은 두 변수 즉 파일명(탑재등록부나 블로크장치파일)과 기발집합을 받는다.

봉사루틴은 다음과 같이 동작한다.

① path_lookup()과 path_walk()를 호출하여 탑재지점경로명을 탐색한다.

이 작업이 끝나면 함수는 경로명에 대응하는 등록부입구점객체의 주소 d를 반환한다.

② 등록부가 파일체계의 탑재위치가 아니면 EINVAL오유코드를 반환한다.

이를 위해 d->mnt->mnt_root가 등록부객체 d의 주소를 포함하는가를 검사한다.

③ 탑재해제할 파일체계가 체계등록부나무에 탑재되지 않았으면 EINVAL오유코드를 반환한다.(어떤 특수파일체계는 탑재지점이 없음을 기억할것이다.) 이를 위해 d->mnt에 대해 check_mnt()함수를 호출하여 검사한다.

④ 사용자가 파일체계를 탑재해제하는데 필요한 권한이 없다면 EPERM 오유코드를 반환한다.

⑤ do_umount()를 호출하여 다음 연산을 수행한다.

ㄱ. 탑재된 파일체계객체의 mnt_sb마당에서 초블로크객체의 주소를 얻는다.

ㄴ. 사용자가 force항목을 통해 탑재해제를 반드시 수행하기를 요청하였다면 umount_begin초블로크연산을 호출하여 진행중인 탑재연산을 중지한다.

ㄷ. 탑재해제할 파일체계가 뿌리파일체계이고 사용자가 뿌리의 분리(detach)를 요청한것이 아니라면 do_remount_sb()를 호출하여 뿌리파일체계를 읽기전용으로 다시 탑재하고 끝낸다.

ㄹ. mount_sem신호기를 쓰기용으로 획득하고 dcache_lock 등록부입구점스핀잠그기를 획득한다.

ㅁ. 탑재된 파일체계가 자식탑재파일체계를 위한 탑재위치를 가지고있지 않거나 사용자가 파일체계의 분리(detach)를 요청하였다면 umount_tree()를 호출하여(모든 자식파일체계와 함께) 파일체계를 탑재해제한다.

ㅂ. mount_sem와 dcache_lock를 해제한다.

6. 경로명탐색

여기서는 VFS가 파일명으로부터 대응하는 i마디를 어떻게 얻는가를 설명한다.

어떤 프로쎄스가 파일을 표현해야 할 때 즉 VFS체계호출인 open(), mkdir(), rename(), stat() 등에 전달해야 할 때 프로쎄스는 파일의 경로명을 전달한다.

이 작업을 수행하는 표준절차는 경로명을 분석하여 여러 파일명의 렬거로 나누는것이다.

마지막을 제외한 파일명은 등록부를 표현한다.

경로명에서 첫번째 문자가 《/》이면 절대경로며 탐색을 current->fs->root (프로쎄스의 뿌리등록부)가 나타내는 등록부에서 시작한다.

그렇지 않으면 경로명은 상대경로이며 탐색을 current->fs->pwd(프로쎄스의 현재등록부)가 나타내는 등록부에서 시작한다.

시작위치등록부의 i마디를 얻었으므로 이 등록부에서 첫번째 이름과 일치하는 입구점을 찾아서 대응하는 i마디를 얻는다.

그리고 이 i마디를 포함하는 등록부파일을 등록부에서 읽은 다음 두번째 이름과 일치하는 입구점을 찾아서 대응하는 i마디를 얻는다.

이 과정을 경로명에 포함된 파일명에 대해 반복한다.

가장 최근에 사용한 등록부입구점객체들을 기억기에 유지하고있는 등록부입구점캐쉬가 이 경로명탐색과정을 매우 빠르게 한다.

앞에서 본것처럼 이 캐쉬의 각 객체는 어떤 등록부의 파일명과 이에 대응하는 i마디를 련결한다.

따라서 많은 경우 경로명해석과정중 중간에 있는 등록부들을 디스크에서 읽지 않아

도 된다.

그렇지만 다음과 같은 Unix와 VFS파일체계의 특성을 반드시 고려해야 하므로 일은 보기처럼 단순하지 않다.

- 프로쩨스가 등록부의 내용을 읽을수 있는 권한이 있는지 확인하기 위해 각 등록부의 접근권한을 검사해야 한다.
- 파일명이 다른 임의의 경로명을 가리키는 기호련결일수도 있다. 이 경우 기호련결이 가리키는 경로명의 모든 구성요소를 분석에 포함해야 한다.
- 기호련결은 원형참조에 도달할수 있다. 핵심부는 이 경우를 방지하기 위해 무한순환이 발생하면 중지한다.
- 파일명은 탑재된 파일체계의 탑재위치일수도 있다. 핵심부는 이 경우를 반드시 검출하여 새 파일체계에서 탐색연산을 계속해야 한다.

경로명탐색은 path_lookup(), path_walk(), npath_release()라는 세 함수를 순서대로 호출하여 실행한다.

path_lookup()은 다음 3가지 변수를 받는다.

name

해석할 파일경로명을 가리키는 지적자

flags

탐색할 파일을 어떻게 접근할것인지 나타내는 기발값.

기발은 뒤에 나오는 《open()체계호출》에서 표 2-17에 있다.

nd

nameidata구조체자료구조의 주소

path_walk()가 경로명탐색연산에 관련된 정보들로 nameidata구조체자료구조를 채운다.

이 구조체의 마당은 표 2-14와 같다.

표 2-14. nameidata 자료구조와 마당

형	마당	설명
struct dentry*	dentry	등록부입구점객체의 주소
struct vfs_mou	mnt	탑재된 파일객체의 주소
struct qstr	last	경로이름의 마지막구성요소 (LOOKUP_PARENT기발이 설정되여있으면 사용)

unsigned int	flags	탐색기발
int	last _type	경로이름의 마지막 구성 요소류형 (LOOKCP_PARENT기발이 설정되여있으면 사용)

dirty와 mnt마당은 경로명에서 마지막으로 해석한 구성요소의 등록부입구점객체와 탑재된 파일체계객체를 가리킨다.

path_walk()가 성공적으로 끝나면 주어진 경로명이 나타내는 파일을 이 두마당이 표현한다.

flags마당은 탐색연산중에 사용하는 기발값을 나타내며 표 2-15와 같다.

표 2-15. 탐색연산의 기발

마 크 로	설 명
LOOKUP_FOLLOW	마지막구성요소가 기호련결이면 해석한다.(따라간다.)
LOOKUP_DIRECTORY	마지막 구성요소가 반드시 등록부여야 한다.
LOOKUP_CONTINU	경로이름에 아직도 해석할 파일이름이 남아있다.(NFS에서만 사용)
LOOKUP_POSITIVE	경로이름이 반드시 존재하는 파일을 나타내야한다.
LOOKUP_PARENT	경로이름의 마지막 구성요소를 포함하는 등록부를 탐색한다.
LOOKUP_NOALT	뿌리등록부모방을 고려하지 않는다. (80x86방식에서 언제나 설정된다.)

path_lookup()함수의 목표는 nameidata자료구조를 초기화하는것으로 다음과 같은 작업을 수행한다.

① dentry마당을 경로명탐색을 시작하는 등록부의 등록부입구점객체주소로 설정한다.

경로명이 상대경로면(《/》으로 시작하지 않으면) 이 마당은 작업등록부(current->fs->pwd)의 등록부입구점을 가리킨다.

그렇지 않으면 프로쎄스의 뿌리등록부(curren->fs->root)의 등록부입구점을 가리

교육성 교육정보쎈터

킨다.

② mnt마당을 경로명탐색을 시작하는 등록부에 대해 탑재된 파일체계의 주소로 설정한다.

경로명이 상대경로인지 절대경로인지에 따라 current->fs->pwdmnt거나 current->fs->rootmnt이다.

③ flags마당을 flags변수값으로 설정한다.

④ LAST_ROOT의 last_type마당을 초기화한다.

path_lookup()이 nameidata를 초기화하고나면 path_walk()함수가 탐색연산을 계속 처리한다.

등록부입구점객체지적자와 경로명의 마지막구성요소에 대한 탑재된 파일체계객체를 nameidata구조체에 저장한다.

또한 nd->dentry와 nd->mnt가 나타내는 객체의 사용계수기를 1증가시켜 path_walk함수가 끝나면 이것을 호출한 함수가 안전하게 사용할수 있게 한다.

호출한 함수가 사용을 끝내면 path_release()를 호출한다. 이 함수는 nameidata자료구조의 주소를 변수로 받아서 nd->dentry와 nd-> mnt의 사용계수기를 감소시킨다.

이제 경로명탐색연산의 핵심인 path_walk()함수를 설명할 차례이다. 이 함수는 해석할 경로명을 가리키는 지적자와 nameidata자료구조의 주소 nd를 변수로 받는다.

현재프로쎄스의 total_link_count를 0으로 초기화하고(뒤에 나오는 기호련결탐색 참고) link_path_walk()를 호출한다.

이 함수는 path_walk()와 같은 두개 변수를 받는다.

리해를 돕기 위해 LOOKUP_PARENT가 설정되여있지 않고 경로명이 기호련결을 포함하지 않은 경우(표준경로명탐색)를 먼저 설명한다.

다음으로 LOOKUP_PARENT가 설정된 경우를 설명한다. 이 류형의 탐색은 등록부입구점을 생성, 삭제, 변경할 때 즉 부모경로명탐색과정에 필요하다.

마지막으로 기호련결을 어떻게 처리하는가를 설명한다.

1) 표준경로명탐색

LOOKUP_PARENT기발이 설정되여있지 않으면 link_path_walk()는 다음 단계를 수행한다.

① lookup_flags국부변수를 nd->flags로 초기화한다.

② 경로명의 첫 구성요소앞에 있는 모든 빗선기호(《/》)를 건너뛴다.

③ 남아있는 경로명이 없으면 0을 반환한다. nameidata자료구조의 dentry와 mnt마당은 원래 경로명에서 마지막으로 해석된 구성요소에 관련한 객체들을 가리킨다.

④ 현재프로쎄스의 서술자의 link_count마당이 정수면 lookup_flags 국부변수의 LOOKUP_FOLLOW기발을 설정한다.(《기호련결탐색》참고)

⑤ 이름을 구성요소로 분할하는 과정을 반복한다.(중간에 있는 빗선기호를 파일명구

분자로 사용한다.) 각 구성요소를 얻을 때마다 함수는 다음을 수행한다.

ㄱ. 마지막으로 해석된 구성요소의 i마디객체의 주소를 nd->dentry->d_inode에서 가져온다.

ㄴ. 마지막으로 해석되여 i마디에 저장된 구성요소의 접근권한이 실행을 허가하는지 검사한다.(Unix에서 실행가능한 등록부만 탐색할수 있다.)

i마디가 독자적인 접근권한메쏘드를 가지고있으면 해당 메쏘드를 실행한다.

그렇지 않으면 vfs_permission()함수를 실행한다. 이 함수는 i마디마당 i_mode와 실행중인 프로쎄스의 권한을 검사한다.

ㄷ. 다음으로 해석할 구성요소를 찾는다. 등록부입구점캐쉬 하쉬표에 사용할 하쉬값을 구성요소이름으로부터 계산한다.

ㄹ. 해석이 끝난 구성요소이름 다음에 뒤따라오는 빗선기호(《/》)들을 건너뛴다.

ㅁ. 해석할 구성요소가 원래 경로명의 마지막것이였으면 6단계로 간다.

ㅂ. 구성요소의 이름이 《.》이면 다음 구성요소에서 계속한다.(《.》은 현재 등록부를 나타내며 따라서 경로명에 아무런 영향을 주지 않는다.)

ㅅ. 구성요소의 이름이 《..》이면 부모등록부로 이동을 시도한다.

· 마지막으로 해석한 등록부가 프로쎄스의 뿌리등록부이면 (nd->dentry가 current->fs->root 와 일치하고 nd->mnt가 current->fs->rootmnt) 다음 구성요소에서 계속한다.

· 마지막으로 해석한 등록부가 탑재된 파일체계의 뿌리등록부이면 (nd->dentry가 nd->mnt->mnt_root 와 일치) nd->mnt를 nd->mnt->mnt_parent로 설정하고 nd->dentry를 nd->mnt-> mnt_mountpoint로 설정하고 ⑤-ㅅ단계에서 다시 시작한다.(같은 탑재위치에 여러 파일체계를 탑재할수 있다고 앞에서 설명하였다.) ·마지막으로 해석한 등록부가 탑재된 파일체계의 뿌리등록부가 아니면 nd->dentry를 nd->dentry->d_parent로 설정하고 다음 구성요소에 대해 계속한다.

ㅇ. 여기에 이르면 구성요소이름은 “.”도 “..”도 아니다. 이제 함수는 등록부입구점캐쉬를 찾는다. 저수준파일체계가 독자적인 d_hash등록부입구점메쏘드를 가지고있으면 함수는 이 메쏘드를 호출하여 ⑤-ㄷ단계에서 계산한 하쉬값을 수정한다.

ㅈ. nd->dentry, 해석할 구성요소이름, 하쉬값, LOOKUP_ CONTINUE기발 등을 변수로 caches_lookup()을 호출한다. 마지막의 LOOKUP_CONTINUE기발은 이것이 경로명의 마지막구성요소가 아님을 나타낸다. 함수는 d_lookup()을 호출하여 구성요소의 등록부입구점객체를 등록부입구점캐쉬에서 찾는다. cached_lookup()이 등록부입구점을 캐쉬에서 찾지 못하면 link_walk_path()는 real_lookup()을 호출하여 등록부를 디스크에서 읽어서 새로운 등록부입구점객체를 생성한다. 어떤 경우든지 이 단계의 끝에서 dentry국부변수는 이 단계에서 해석할 구성요소이름의 등록부입구점객체를 가리킨다.

ㅊ. 방금 해석한 구성요소(dentry국부변수)가 파일체계에 대한 탑재지점으로 사용되는 등록부를 나타내는가를 검사한다.(dentry->d_mounted가 1로 설정되여있는 경우)이 경우 dentry와 nd->mnt를 변수로 lookup_mnt()를 호출하여 탑재된 자식파일체계객체의 주소 mounted로 설정한다.

그리고 전체 단계를 다시 반복한다(여러 파일체계를 같은 탑재위치에 탑재할수 있다.)

ㅋ. dentry->d_inode i마디객체가 독자적인 follow_link메쏘드를 가지고있는가를 검사한다. 그렇다면 구성요소는 기호련결이다. 《기호련결탐색》에서 기호련결을 다룬다.

ㅌ. dentry가 등록부의 등록부입구점객체를 가리키는가를 검사한다.(entry->d_inode->I_op->lookup메쏘드가 정의됨) 그렇지 않으면 구성요소는 원래 경로명의 중간에 있는데 dentry가 등록부가 아니므로 ENOTDIR오유를 반환한다.

⑥ 자기의 원래 경로명에서 마지막을 제외한 모든 구성요소를 해석하였다. 경로명에 뒤따라오는 빗선기호가 있으면 lookup_flags국부변수에서 LOOKUP_FOLLOW와 LOOKUP_DIRECTORY를 설정하여 마지막구성요소를 등록부명으로 해석하도록 한다.

⑦ lookup_flags변수의 LOOKUP_PARENT기발을 검사한다.

여기서는 이 기발이 0이라고 가정하며 1인 경우는 뒤에서 다룬다.

⑧ 마지막구성요소의 이름이 《.》이면 실행을 완료하고 0(오유없음)을 반환한다. nd가 가리키는 nameidata구조의 dentry 와 mnt마당은 경로명의 마지막직전구성요소가 나타내는 객체를 가리킨다(《.》은 경로명에서 아무런 영향을 주지 않는다.)

⑨ 마지막구성요소의 이름이 《..》이면 부모등록부로 이동을 시작한다.

ㄱ.마지막으로 해석한 등록부가 프로쎄스의 뿌리등록부이면 (nd->dentry가 current->fs->mnt와 일치하고 nd->mnt 가 current->fs->rootmnt) 실행을 완료하고 0(오유없음)을 반환한다.

ㄴ. 마지막으로 해석한 등록부가 탑재된 파일체계의 뿌리등록부이면(nd->dentry가 nd->mnt->mnt_root와 일치) nd->mnt를 nd->mnt->mnt_parent로 설정하고 nd->dentry를nd->mnt->mnt_mountpoint로 설정하고 ⑤-ㅊ단계에서 다시 시작한다.

ㄷ. 마지막으로 해석한 등록부가 탑재된 파일체계의 뿌리등록부가 아니면 nd->dentry를 nd->dentry->d_parent로 설정하고 실행을 완료하고 0(오유 없음)을 반환한다.

nd->dentry와 nd->mnt는 경로명의 마지막직전구성요소가 나타내는 객체를 가리킨다.

마지막구성요소의 이름은 《.》도 《..》도 아니다. 이제 함수는 등록부입구점캐쉬를 찾는다.

저수준파일체계가 독자적인 d_hash등록부입구점메쏘드를 가지고있으면 함수는 이

메쏘드를 호출하여 ⑤-ㄷ단계에서 계산한 하쉬값을 수정한다.

nd->dentry, 해석할 구성요소이름, 하쉬값 그리고 기발없음 등을 변수로 cached_lookup()을 호출한다.(이것이 경로명의 마지막구성요소이므로 LOOKUP_CONTINUE가 설정되지 않는다.)

cached_lookup()이 등록부입구점을 캐쉬에서 찾지 못하면 real_lookup()을 호출하여 디스크에서 등록부를 읽어서 새로운 등록부입구점객체를 생성한다.

어떤 경우든지 이 단계의 끝에서 dentry국부변수는 이 단계에서 해석할 구성요소이름의 등록부입구점객체를 가리킨다고 가정할수 있다.

방금 해석한 구성요소(dentry국부변수)가 파일체계에 대한 탑재위치로 사용되는 등록부를 나타내는지 검사한다.(dentry-> d_mounted가 1로 설정되여있는 경우)

이 경우 dentry와 nd->mnt를 변수로 lookup_mnt()를 호출하여 탑재된 자식파일체계 객체의 주소 mounted를 얻는다.

다음으로 dentry를 mounted->mnt_root로 설정하고 nd->mnt를 mounted로 설정한다.

그리고 전체 단계를 다시 반복한다.(여러 파일체계를 같은 탑재지점에 탑재할수 있다.)

LOOKUP_FOLLOW기발이 설정되여있고 dentry->d_indoe i마디객체가 독자적인 follow_link메쏘드를 가지고있는지 검사한다.

둘다 만족하면 구성요소는 기호련결이다.(《기호련결탐색》에서 기호련결을 다룬다.) nd->dentry를 dentry국부변수에 저장된 값으로 설정한다. 이 등록부입구점객체가 탐색연산의 최종결과이다.

nd->dentry->d_inode가 NULL인가를 검사한다. 이것은 등록부입구점객체에 대응하는 i마디가 없는 경우로, 대부분 존재하지 않는 파일을 경로명이 가리키는 경우이다.

lookup_flags에 LOOKUP_POSITIVE나 LOOKUP_DIRECTORY가 설정되여있다면 ENOENT오유코드를 반환하면서 완료한다.

그렇지 않으면 0(오유 없음)을 반환하면서 완료한다. nd->dentry는 탐색연산에 의해 생성된 negative등록부입구점객체를 가리킨다.

경로명의 마지막구성요소에 대응하는 I마디가 존재한다.

lookup_flags에 LOOKUP_DIRECTORY기발이 설정되여있으면 inode가 독자적인 탐색메쏘드를 가지는가 즉 등록부인가를 검사한다.

등록부가 아니면 ENOTDIR오유코드를 반환하여 완료한다.

0(오유 없음)을 반환하면서 완료한다. nd->detnry와 nd->mnt는 경로명의 마지막구성요소를 가리킨다.

2) 부모경로명탐색

많은 경우 탐색연산의 실제목표는 경로명의 마지막구성요소가 아닌 마지막 직전의

구성요소이다.

례를 들어 파일을 생성할 때 마지막구성요소는 아직 존재하지 않는 파일명을 나타내고 경로명의 나머지부분이 새로운 련결을 삽입할 등록부를 나타낸다.

따라서 탐색연산은 마지막 직전구성요소의 등록부입구점객체를 얻어야 한다.

또 다른 실례로 경로명 /foo/bar 로 표현하는 파일을 련결해제하는것은 bar를 등록부foo에서 삭제하는것이다. 따라서 핵심부는 bar가 아닌 파일등록부인 foo에 접근해야 한다.

탐색연산이 경로명의 마지막 구성요소가 아닌 마지막구성요소를 포함하는 등록부를 찾아야할 때 LOOKUP_PARENT기발을 사용할수 있다.

LOOKUP_PARENT기발이 설정되면 path_walk()함수는 nameidata 자료구조의 last와 last_type마당도 설정한다. last마당은 경로명의 마지막구성요소를 저장한다.

last_type마당은 마지막구성요소의 류형을 저장한다. last_type마당의 값은 표 2-16에 렬거한 값중 하나이다.

표 2-16. nameidata자료구조의 last_type마당값

값	설명
LAST_NORM	마지막구성요소는 일반파일명이다.
LAST_ROOT	마지막구성요소는 《/》이다.(즉 전체 경로명이 《/》이다.)
LAST_DOT	마지막구성요소는 《.》이다.
LAST_DOTDOT	마지막구성요소는《..》이다.
LAST_BIND	마지막구성요소는 특수파일체계를 가리키는 기호련결이다.
O_NONBLOCK	파일에 대하여 어떤 체계호출도 차단되지 않음
O_NDELAY	O_NONBLOCK과 같음
O_SYNC	동기적인 쓰기 (물리적인 쓰기를 완료할 때까지 차단됨)
FASYNC	신호를 통해 비동기적으로 입출력알림
O_DIRECT	직접I/O전송(핵심부완충을 하지 않음)
O_LARGEFTLE	큰 파일(2GB보다 큰 경우)
O_DIRECTORY	파일이 등록부가 아니면 실패함
O_NOFOLLOW	경로이름에서 기호련결을 따라가지 않음

LAST_ROOT기발은 전체적인 경로명탐색연산이 시작할 때 path_lookup()이 설정하는 초기값이다.(《경로명탐색》에서 시작부분에 있는 설명을 참고)

전체 경로명이 《/》으로 판명되면 핵심부는 last_type마당의 초기값을 변경하지 않는다.

LAST_BIND기발은 특수파일체계에 있는 기호련결의 follow_link i마디객체의 메

쏘드에 의해 설정된다.

last_type마당의 다른 값들은 LOOKUP_PARENT기발이 설정되였을 때 link_path_walk()에 의해 설정된다.

이 경우 함수는 앞의 7단계까지 같은 단계를 실행한다. 그러나 7단계 이후부터 마지막구성요소에 대한 탐색연산은 다르게 실행한다.

① nd->last를 마지막구성요소의 이름으로 설정한다.

② nd->last_type을 LAST_NORM으로 초기화한다.

③ 마지막구성요소의 이름이 《.》이면 nd->last_type를 LAST_DOT로 설정한다.

④ 마지막구성요소의 이름이 《..》이면 nd->last_type를 LAST_DOTDOT로 설정한다.

⑤ 0(오유 없음)을 반환하며 완료한다.

여기서 보는것처럼 마지막구성요소는 해석하지 않는다. 따라서 함수가 완료할 때 nameidata자료구조의 dentry와 mnt마당은 마지막구성요소를 포함하고있는 등록부를 나타내는 객체를 가리키게 된다.

3) 기호련결탐색

기호련결은 다른 파일의 경로명을 저장하고있는 정규파일이라는 사실을 기억할것이다.

저장하고있는 경로명도 다시 기호련결을 포함할수 있으며 이런 경우도 핵심부가 해석해야 한다.

례를 들어 /foo/bar가 ../dir를 가리키는 기호련결이라면 핵심부는 경로명 /foo/bar/file을 해석하여 /dir/file을 참조하게 해야 한다.

이 실례에서 핵심부는 두가지 탐색연산을 실행해야 한다.

첫번째는 /foo/bar를 해석하는것이다.

핵심부는 bar가 기호련결의 이름이라는 사실을 발견하면 그 내용을 읽어 다른 경로명으로 해석해야 한다.

두번째 경로명해석은 첫번째 연산결과 도달한 등록부에서 시작하며 기호련결경로명이 마지막구성요소를 해석할 때까지 실행한다.

이어서 두번째탐색연산이 도달한 등록부입구점으로부터, 원래 탐색연산의 실행이 원래 경로명의 기호련결이후 구성요소들을 해석한다.

이런 실행순서를 더 복잡하게 하는것은 기호련결에 포함된 경로명이 다시 기호련결을 포함할수 있다는 점이다.

이쯤하면 아마도 기호련결을 해석하는 핵심부코드가 무척 리해하기 어려울것이라고 짐작되지만 재귀호출(recursion)이 있음으로 하여 코드는 단순한편이다.

그렇지만 조종하지 않은 재귀호출은 근본적으로 위험하다.

례를 들어 기호련결이 자기자신을 가리킨다고 하자.

이와 같은 기호련결을 포함하는 경로명을 해석하는것은 끝없는 재귀호출에 이를것이

고 결국 핵심부탄창기억기자리넘침에 이르게 될것이다.

이 문제를 해결하기 위해 현재프로쎄스의 서술자에 있는 link_count마당을 사용한다.

이 마당은 각 재귀호출실행직전에 증가하고 실행이후에 감소한다. 마당의 값이 5에 이르면 전체 탐색연산이 완료되며 오유코드를 반환한다.

따라서 기호련결의 최대값은 5이다.

또 현재프로쎄스의 서술자의 total_link_count마당이 원래 탐색연산에서 얼마나 많은 기호련결을 거쳤는지 파악한다.

이 계수기가 40에 이르면 탐색연산을 끝낸다. 이 계수기가 없다면 악의적인 사용자가 아주 많은 련속된 기호련결을 포함하는 나쁜 경로명을 생성하여 핵심부가 아주 오래동안 탐색연산을 계속하도록 할수 있다.

탐색코드는 기본적으로 다음과 같이 동작한다.

link_path_walk()함수가 경로명의 구성요소에 대응하는 등록부입구점객체를 가져오고 대응하는 i마디객체가 독자적인 follow_link메쏘드를 가지는지 검사한다.(《표준경로명탐색》의 ⑤-ㅋ참고) 메쏘드를 가진다면 i마디는 원래의 경로명의 탐색연산을 계속 진행하기 전에 해석해야 하는 기호련결을 포함하고있는것이다.

이 경우 link_path_walk()함수는 기호련결의 등록부입구점객체의 주소와 nameidata자료구조의 주소를 변수로 do_follow_link()를 호출한다.

이어서 do_follow_link()는 다음의 단계를 실행한다.

① current->link_count가 5보다 작은지 검사한다. 그렇지 않으면 ELOOP 오유코드를 반환한다.

② current->total_link_count가 40보다 작은지 검사한다. 그렇지 않으면 ELOOP오유코드를 반환한다.

③ current->need_resched기발이 설정되여있으면 schedule()을 호출하여 현재프로쎄스를 선취(preempt)할 기회를 준다.

④ cureent->link_count와 current->total_link_count를 1씩 증가시킨다.

⑤ 해석할 기호련결에 대응하는 i마디객체의 접근시간을 갱신한다.

⑥ 등록부입구점객체의 주소와 nameidata자료구조의 주소를 변수로 follow_link메쏘드를 호출한다.

⑦ current->link_count마당을 감소시킨다.

⑧ follow_link메쏘드가 반환한

⑨ 오유코드를 반환한다.(0이면 오유가 없는것이다.)

⑩ follow_link메쏘드는 파일체계의 독자적인 함수로 디스크의 기호련결에 저장된 경로명을 읽어야 한다.

기호련결의 경로명을 완충기에 채운 다음 대부분의 follow_link메쏘드는 vfs_follow_link()함수를 호출하고 이 함수의 결과를 반환한다. vfs_follow_link()는

다음과 같은 작업을 수행한다.

① 기호련결경로명의 첫번째문자가 빗선기호인지 검사한다. 빗선기호면 현재프로쎄스의 뿌리등록부를 가리키도록 nameidata자료구조의 dentry와 mnt마당을 설정한다.

② nameidata자료구조를 변수로 link_path_walk()를 호출하여 기호련결경로명을 해석한다.

③ link_path_walk()에서 받은값을 반환한다.

do_follow_link()가 완전히 끝나면 이 함수는 기호련결이 가리키는 등록부입구점객체의 주소를 link_path_walk()에 반환한다.

link_path_walk()는 이 주소를 entry국부변수에 대입하고 다음 단계로 넘어간다.

7. VFS체계호출실현

이절의 처음에서 보았던 실례를 다시 보자. 사용자가 MS-DOS파일 /floppy/TEST를 Ext2파일 /tmp/test에 복사하는 쉘명령을 실행한다.

쉘은 cp와 같은 외부프로그람을 실행하고 이 프로그람은 다음과 같은 코드를 실행한다고 가정한다.

```
inf = open(  /floppy/TEST  , O_RDONLY, 0);
outf = open(“/tmp/test”,O_WRONLY|O_CREAT|O_TRUNC, 0600);
do{
        len = read(inf, buf, 4096);
        write(outf, buf, len);
}while(len);
close(outf);
close(inf);
```

사실 실제 cp프로그람의 코드는 매 체계호출에서 반환하는 오유코드를 검사해야 하기때문에 더 복잡하다.

이 실례는 copy연산의 정상적인 동작에만 초점을 맞춘것이다.

1) open()체계호출

open()체계호출은 sys_open()함수가 처리하는데 이 함수는 열려는 파일경로명 filename과 접근방식기발 flags 그리고 파일을 생성해야 하는 경우에는 접근권한비트마스크 mode를 변수로 받는다.

체계호출이 성공하는 경우 파일서술자 즉 파일객체에 대한 지적자배렬인 current->files->fd의 색인값을 반환한다.

실패하는 경우 1을 반환한다.

실례에서는 open()을 두번 호출한다.

처음에 /floppy/TEST읽기 위해 열고 (O_RDONLY기발) 그 다음 /tmp/test를

쓰기 위해 연다.(O_WRONLY 기발.)

/tmp/test가 존재하지 않는다면 파일을 생성하고(O_CREAT기발) 소유자에 대해 배타적인 읽기, 쓰기접근권한만 허용한다.(세번째 변수의 8진수 0600)

파일이 이미 있다면 파일내용을 모두 지우고 다시 쓰기를 시작한다. (O_TRUNC기발) 표 2-17은 open()체계호출의 모든 기발을 보여준다.

표 2-17. open()체계호출기발

기발이름	설명
O_RDONLY	읽기용으로 파일열기
O_WRONLY	쓰기용으로 파일열기
O_RDWR	읽기/쓰기용으로 파일열기
O_CREAT	파일이 존재하지 않으면 생성
O_EXCL	O_CREAT와 함께 사용하며 파일이 이미 존재하면 실패
O_NOCTTY	파일을 조종말단으로 간주하지 않음
O_TRUNC	파일내용을 자름 (기존의 모든 내용을 삭제함)
O_APPEND	언제나 파일의 끝에 쓰기를 수행함
O_NONBLOCK	파일에 대해서 어떤 체계호출도 차단되지 않음
O_NDELAY	O_NONBLOCK과 같음
O_SYNC	동기적인 쓰기(물리적인 쓰기를 완료할 때까지 차단됨)
FASYNC	신호를 통해 비동기적으로 입출력알림
O_DIRECT	직접I/O전송 (핵심부완충을 하지 않음)
O_LARCEFILE	큰 파일 (2GB 보다 큰 경우)
O_DIRECTORY	파일이 등록부가 아니면 실패함
O_NOFOLLOW	경로이름에서 기호련결을 따라가지 않음

이제 sys_open()함수의 동작을 살펴보자. 함수는 다음과 같은 과정을 수행한다.

① getname()을 호출하고 프로쎄스주소공간에서 파일경로명을 읽는다.

② get_unused_fd()를 호출하여 current->files->fd에서 빈공간을 찾는다. 대응하는 색인값(새로운 파일서술자)를 fd국부변수에 저장한다.

③ 경로명, 접근방식기발 그리고 접근권한비트마스크를 변수로 file_open()함수를 호출한다. 이 함수는 다음과 같은 과정을 거친다.

ㄱ. 접근방식기발을 namei_flags에 복사하는데 접근방식기발인 O_RDONLY, O_WRONLY, O_RDWR를 경로명탐색함수가 요구하는 형식으로 변환한다.(《경로명탐색》 참고)

ㄴ. 경로명, 변경된 접근방식기발, nameidata자료구조를 변수로 open_namei()함수를 호출한다. 함수는 다음과 같이 탐색연산을 수행한다.

・접근방식기발에 O_CREAT가 설정되여있지 않으면 LOOKUP_PARENT기발을 설정하지 않고 탐색연산을 시작한다. O_NOFOLLOW기발이 설정되여있지 않을 때에만 LOOKUP_FOLLOW기발을 설정한다. O_DIRECTORY기발이 설정되여있을 때에만 LOOKUP_DIRECTORY기발을 설정한다.

・접근방식기발에 O_CREAT기발이 설정되여있으면 LOOKUP_ PARENT기발을 설정하여 탐색연산을 시작한다.

path_walk()에서 성공적으로 돌아오면 요청한 파일이 이미 존재하는지 검사한다.

존재하지 않는다면 부모 i마디의 creat메쏘드를 호출하여 새로운 디스크 i마디를 할당한다.

open_namei()함수는 탐색연산이 찾은 파일에 대해 여러가지 보안검사를 실행한다.

례를 들어 찾은 등록부입구점객체에 대응하는 i마디가 정말 존재하는지, 정규파일인지, 현재프로쎄스가 접근방식기발에 따라 이 파일을 사용할수 있는지 등을 검사한다.

쓰기용으로 파일을 여는 경우 이미 다른 프로쎄스가 파일에 잠그기를 걸었는가를 검사한다.

ㄷ. 접근방식기발, 등록부입구점객체의 주소, 탐색연산이 찾은 탑재된 파일체계객체를 변수로 dentry_open()함수를 호출한다. 이 함수는 다음을 수행한다.

・새로운 파일객체를 할당한다.

・파일객체의 f_flags마당과 f_mode마당을 open()체계호출에 전달된 접근방식기발에 따라 초기화한다.

・f_dentry와 fl_vfsmnt마당을 변수로 받은 등록부입구점객체의 주소와 탑재된 파일체계객체에 따라 초기화한다.

・f_op마당을 대응하는 i마디객체의 i_fop마당의 내용으로 설정한다.

이것으로 이후 파일연산을 위한 메쏘드설정은 끝났다.

・파일객체를 파일체계의 초블로크의 s_files마당이 가리키는 열린 파일의 목록에 삽입한다.

・O_DIRECT마당이 설정되여있으면 디스크접근완충기를 미리 할당한다.

・파일연산의 open메쏘드가 정의되여있으면 호출한다.

ㄹ. 파일객체의 주소를 반환한다.

current->files->fd[fd]값을 dentry_open()이 반환하는 파일객체의 주소로 설정한다.

fd를 반환한다.

2) read()와 write()체계호출

다시 cp실례코드로 돌아가자.

두 open()체계호출은 파일서술자를 하나씩 반환하여 각각 inf와 outf변수에 저장한다.

프로그람은 순환을 돌기 시작한다. 매 반복과정에서 /floppy/TEST파일의 일부를 지역완충기에 복사하고(read()체계호출) 지역완충기의 자료를 /tmp/test에 기록한다. (write()체계호출)

read()와 write()체계호출은 아주 류사하다.

둘다 다음과 같은 세 변수를 요구한다.

즉 파일서술자 fd, 기억기령역(전송할 자료를 포함한 완충기)주소 buf 그리고 얼마나 많은 완충기를 전송해야 하는가를 나타내는 수자 count이다.

물론 read()는 자료를 파일에서 완충기로 전송하고 write()는 반대로 동작한다.

두 체계호출은 성공적으로 전송한 바이트수를 반환하거나 오유조건을 나타내기 위해 1을 반환한다.

count보다 작은 반환값이 오유가 발생하였다는 사실을 의미하지는 않는다.

핵심부는 요청한 바이트를 모두 전송하기 전에도 체계호출을 완료할수 있으므로 사용자응용프로그람은 반드시 반환값을 검사하여 필요하다면 체계호출을 다시 호출해야 한다.

대개 관이나 말단장치에서 읽는 경우 파일의 끝(EOF)을 지나 읽는 경우 또는 신호가 체계호출을 새치기한 경우에 작은 값을 반환한다.

파일의 끝(EOF)조건은 read()에서 반환하는 0값으로 쉽게 판별할수 있다.

이 조건은 read()가 아무런 자료도 읽기전에 신호에 의해 새치기한 경우 오유가 발생하므로 신호에 의한 비정상완료와 혼동되지 않는다.

read와 write연산은 언제나 현재파일지적자(파일객체의 f_pos마당)가 나타내는 파일편위에서 시작한다.

두 체계호출은 전송한 바이트수만큼 파일객체에 값을 더해 갱신한다.

요약하면 sys_read()(read()의 봉사루틴)와 sys_write(write()의 봉사루틴)은 거의 같은 과정을 거친다.

① fget()를 호출하여 fd로부터 대응하는 파일객체의 주소 file을 얻고 사용계수기 file->f_count를 증가시킨다.

② 요청한 접근을 허용하는지 file->f_mode기발을 검사한다.(read 또는 write 연산)

③ locks_verify_area()를 호출하여 접근할 파일범위에 대해 《강제적열쇠잠그기(mandatory lock)》가 존재하는가를 검사한다.(《파일잠그기》참고)

④ 파일을 전송하기 위해 file->f_op->read또는 file->f_op->write 를 실행한

다. 두 함수는 실제로 전송한 바이트수를 반환한다. 부가적으로 파일지적자로 갱신한다.

⑤ fput()를 호출하여 사용계수기 file->f_count를 감소시킨다.

⑥ 실제로 전송된 바이트수를 반환한다.

3) close()체계호출

실레코드의 순환은 read()체계호출이 0을 반환하면 끝난다.

즉 /floppy/TEST 의 모든 바이트를 /tmp/test로 복사하면 끝난다.

이제 복사연산을 완료했으므로 프로그람은 열린 파일을 닫을수 있다.

close()체계호출은 닫으려는 파일의 서술자의 fd를 변수로 받는다.

sys_close()봉사루틴은 다음과 같은 연산을 수행한다.

① current->files->fd[fd]에 저장한 파일객체주소를 얻는다. NULL이라면 오유코드를 반환한다.

② current->files->fd[fd]를 NULL로 설정한다. 이에 대응하는 비트를 open_fds와 current->files의 close_on_exec마당에서 지워서 파일서술자 fd를 해제한다.

③ filp_close()를 호출하여 다음과 같은 연산을 수행한다.

ㄱ. 파일연산의 flush메쏘드가 정의되여있다면 이것을 실행한다.

ㄴ. 파일에 대해 모든 강제적잠그기를 해제한다.

ㄷ. fput()를 실행하여 파일객체를 해제한다.

④ flush메쏘드의 오유코드를 반환한다.(일반적으로 0이다.)

8. 파일잠그기

프로세스 두개이상이 한 파일에 접근하는 일이 가능한 경우 동기화(synchronization)문제가 발생한다.

두 프로세스가 같은 파일위치에 쓰기를 시도하면 어떻게 되는가? 혹은 한 프로세스가 쓰기작업중인데 다른 프로세스가 같은 위치에서 읽기작업을 수행하면 어떻게 될것인가?

전통적인 Unix체계에서는 같은 파일위치에 동시에 접근하는 일은 예측할수 없는 결과를 낳았다. 그렇지만 Unix체계는 프로세스가 파일령역에 열쇠를 걸수 있도록 하여 동시접근을 쉽게 피할수 있도록 하였다.

POSIX표준은 fcntl() 체계호출을 사용한 파일잠그기기구를 요구한다.

이 체계호출은 임의의 파일령역(1B까지도) 또는 (앞으로 뒤에 덧붙일 자료를 포함한) 전체 파일에 열쇠를 걸수 있다.

프로세스가 파일의 일부분에 열쇠를 걸수 있기때문에 같은 파일내에서도 열쇠를 여러개 사용할수 있다.

이 종류의 잠그기는 잠그기를 무시하는 다른 프로세스를 막을수 없다.

코드의 《림계령역(critical region)》과 마찬가지로 이런 잠그기를 《권고적

(advisory)》이라고 하는데 다른 프로쎄스가 파일에 접근하기 전에 잠그기의 존재를 검사하지 않으면 정상적으로 동작할수 없기때문이다.

따라서 POSIX 의 잠그기를 《권고적잠그기(advisory lock)》이라고 부르기도 한다.

전통적인 BSD계렬은 flock()체계호출을 사용하여 권고적잠그기를 실현한다.

이 체계호출은 프로쎄스의 잠그기를 파일의 일부령역이 아닌 전체 파일에 대해서만 허가한다.

전통적인 System V계렬체계는 lockf()체계호출을 제공하는데 이것은 단지 fcntl()에 대한 대면부이다.

더 중요한것은 System V Release3(SVR3)에서 강제적잠그기(mandatory lock)를 도입하였다는 사실이다.

open(), read() 그리고 write()체계호출을 실행하면 핵심부는 접근하려고 하는 파일에 대해 강제적잠그기를 침범하지 않았는지 검사한다.

따라서 서로 협력하지 않는 프로쎄스들도 강제적잠그기를 지키게 된다.

파일에 대해 강제적잠그기를 사용하도록 표시하기 위해 set_group비트(SGID)를 설정하고 그룹실행허가비트를 0으로 설정한다.

그룹실행비트가 0으로 되여있으면 set_group비트는 의미가 없으며 핵심부는 이 조합을 권고적잠그기가 아닌 강제적잠그기를 사용하라는 암시로 해석한다.

프로쎄스가 권고적잠그기를 사용하든 강제적잠그기를 사용하든, 공유읽기잠그기와 배타적쓰기잠그기를 사용할수 있다.

많은 프로쎄스가 같은 파일 령역에 대해서 읽기잠그기를 가질수 있으나 한 순간에 한 프로쎄스만 쓰기잠그기를 가질수 있다.

그리고 다른 프로쎄스가 같은 파일령역에 대해 읽기잠그기를 가지고있을 때 쓰기잠그기를 얻을수 없고 그 반대 경우도 마찬가지이다.(표 2-18)

표 2-18. 잠그기 허용여부

현재잠그기	허용여부	
	읽기	쓰기
잠그기가 없음	예	예
읽기 잠그기	예	아니
쓰기 잠그기	아니	아니

1) Linux파일잠그기

Linux는 모든 종류의 파일잠그기를 지원한다. 권고적잠그기, 강제적잠그기를 제공하며 fcntl(), flock(), lockf()체계호출을 모두 지원한다.

그러나 lockf()체계호출은 단지 래퍼(wrapper)루틴이므로 여기서 설명하지 않는다.

fcntl()의 강제적잠그기는 mount()체계호출의 MS_MANDLOCK기발 (mand항목)을 사용하여 파일체계단위로 활성화/비활성화할수 있다.

기본적으로는 강제적잠그기를 활성화하지 않는다. 이 경우 fcntl()은 권고적잠그기를 생성한다.

기발을 설정한 경우 파일의 set_group 비트가 설정되여있고 그룹실행비트가 설정되여있지 않으면 fcntl()은 강제적잠그기를 생성한다.

그렇지 않으면 잠그기를 생성한다.

이전 판본의 Linux에서 flock()체계호출은 MS_MANDLOCK탑재 기발과 관계없이 권고적잠그기를 생성하였다.

이것은 모든 Unix계렬 조작체계의 기본적인 동작방식이다.

그러나 Linux 2.6에서 flock()의 강제적잠그기라는 특수한 종류의 잠그기를 추가하였다.

이것은 일부 전용망파일체계실현을 제대로 지원하려고 추가한것이다.

이 잠그기는 공유방식 강제적잠그기라고도 불리운다. 잠그기를 설정하면 다른 프로세스는 이 잠그기의 접근방식과 충돌할수 있는 파일을 열지 못한다.

이 기능을 사용하면 원천코드의 호환성이 떨어지므로 Unix응용프로그람개발에 사용하지 않는것이 좋다.

또한 Linux 2.6에서는 《lease》라는 또 다른 flock()기반의 강제적인 잠그기를 해제하는 기능을 추가하였다.

lease로 보호한 파일을 다른 프로세스가 열려고 하면 다른 경우와 마찬가지로 차단되지만 잠그기를 소유한 프로세스가 신호를 받는다.

신호를 받은 프로세스는 먼저 내용의 일관성을 유지하기 위해 디스크의 파일을 갱신하고 잠그기를 풀어야 한다.

소유한 프로세스가 체계에 정의된 시간간격(초단위수를 /proc/sys/fs/lease-break-time에 기록하면 되며 보통 43s이다)동안 잠그기를 풀지 않으면 핵심부가 lease를 자동적으로 제거하고 차단된 프로세스의 실행을 허가한다.

read(), write()체계호출내부에서의 검사외에도 핵심부는 파일내용을 변경할 가능성이 있는 모든 체계호출을 처리할 때 강제적잠그기의 여부를 검사한다.

례를 들어 파일에 강제적잠그기가 존재하는 경우 O_TRUNC기발을 설정하여 open()체계호출을 실행하면 실패한다.

fcntl()이 만드는 잠그기는 FL_POSIX류형이지만 flock()가 만드는 잠그기는 FL_LOCK, LOCK_MAND(공유방식잠그기의 경우), FL_LEASE(lease의 경우)류형이다.

fcntl()을 통해 생성한 잠그기는 flock()를 통해 생성한 잠그기와 안전하게 공존할수 있으면 서로에 어떤 영향도 주지 않는다.

따라서 fcntl()로 잠그기한 파일은 flock()로는 잠그기가 걸리지 않은것처럼 보이며 반대경우도 마찬가지다.

다음 부분에서 핵심부가 파일잠그기를 처리하기 위해 사용하는 주요 자료구조를 설명한다.

그 다음 두 부분은 주요잠그기류형인 FL_POSIX와 FL_FLOCK의 차이점을 본다.

2) 파일잠그기자료구조

file_lock자료구조는 파일잠그기를 나타낸다.

자료구조마당은 표 2-19에 있다. 모든 file_lock자료구조는 2중련결목록으로 구성된다.

file_lock_list가 첫번째 항목의 주소를 나타내며 fl_nextlink, fl_prevlink 마당이 목록에서 린접한 항목의 주소를 저장한다.

표 2-19. file_lock자료구조와 마당

형	마당	설명
struct file_ lock*	fl_ next	i마디목록의 다음 요소
struct list_ head	fl_ link	대역목록 지시자
struct list_ head	fl_ block	프로쎄스목록 지시자
struct flies_ struct*	fl_ owner	소유자의 files_ struct
unsegnde int	fl_ pid	프로쎄스소유자의 PID
wait_ qucue_ head_ t*	fl_ wait	차단된 프로쎄스의 대기렬
struct file*	fl_file	파일객체지시자
unsigned char	fl_flage	잠그기기발
unsignedchar	fl_type	잠그기류형
loff_t	fl_start	잠그기상태령역 시작편위
loff_t	fl_end	잠그기상태령역 끝편위
void(*)(struct file_lock*)	fl_notify	잠그기가 해제되였을 때 호출하는 되돌이호출(Call-Back)함수
void(*)(struct file_lock*)	fl_inscrt	잠그기sj 삽입되였을 때 호출하는 되돌이호출(Call-Back)함수
void(*)(struct file_lock*)	fl_remove	잠그기가 제거되였을 때 호출하는 되돌이호출(Call-Back)함수
struct fasyne_struct*	fl_fasync	lease중단을 알리는데 사용
union	fl_u	파일체계별 정보

디스크의 같은 파일을 가리키는 모든 file_lock자료구조는 단순련결목록을 구성한다.

i마디객체의 i_flock마당이 첫번째 항목을 가리키고 file_lock구조체의 fl_next마당이 목록의 다음 항목을 가리킨다.

프로쎄스가 권고적잠그기 또는 강제적잠그기를 얻으려고 할 때 같은 파일령역에 대해 이전에 할당한 잠그기가 해제될 때까지 프로쎄스가 보류(suspended)될수 있다.

같은 잠그기에 대해 잠들어있는(sleeping) 모든 프로쎄스는 대기렬(wait queue)에 들어간다.

file_lock구조체의 fl_wait 마당이 대기행렬의 머리부를 가리킨다.

그리고 어떤 파일잠그기에 대해 잠들어있는 모든 프로쎄스를 원형2중련결목록에 삽입한다.

목록의 머리부는 허수아비요소로 block_list변수에 저장되여있으며 file_lock 자료구조의 fl_block마당에 목록에서 린접한 요소의 지적자를 저장한다.

3) FL_LOCK잠그기

FL_LOCK는 언제나 파일객체와 련관되며 따라서 특정프로쎄스(또는 동일한 열린 파일을 공유하는 복제프로쎄스들)가 관리한다.

잠그기를 요청하여 허가받으면 핵심부는 이 프로쎄스가 같은 파일객체에 대해 가지고있는 모든 다른 잠그기를 대체한다.

이런 경우는 프로쎄스가 이미 소유한 읽기잠그기를 쓰기잠그기로 바꾸려고 하거나 그 반대 경우에 발생한다.

그리고 fput()함수로 파일객체를 해제하면 이 파일객체를 참조하는 모든 FL_LOCK잠그기가 해제된다.

그렇지만 같은 파일(inode)에 대해 다른 프로쎄스가 설정한 FL_LOCK 읽기잠그기가 있을수 있는데 이것들은 활성상태로 남는다.

flock()체계호출은 변수 두개를 받는다.

잠그기하려는 파일의 파일서술자 fd와 잠그기연산을 지정하는 cmd이다.

cmd변수값중에서 LOCK_SH는 읽기를 위한 공유잠그기를, LOCK_EX는 쓰기를 위한 배타적잠그기를 얻고 LOCK_UN은 잠그기를 해제한다.

만약 LOCK_NB값을 LOCK_SH 또는 LOCK_EX에 론리합 OR로 추가하면 체계호출은 차단되지 않는다.

만약 즉시 잠그기를 얻을수 없다면 체계호출은 오유코드를 반환한다.

기억할 점은 파일안에서 령역을 지정할수 없다는 사실이다.

잠그기는 언제나 파일전체에 적용된다.

sys_flock()봉사루틴은 다음과 같은 단계로 실행한다.

① fd가 유효한 파일서술자인지 검사한다. 그렇지 않으면 오유코드를 반환한다. 대

응하는 파일객체의 주소를 얻는다.

② 프로쎄스가 권고적잠그기를 획득하려는 경우 프로쎄스가 열린 파일에 대한 읽기나 쓰기권한(또는 둘다)이 있는지 검사한다. 그렇지 않으면 오유코드를 반환한다.

③ flock_lock_file()을 호출한다. 파일객체지적자 filp, 요청하는 잠그기연산의 류형 type, wait기발을 변수로 전달한다. 마지막 변수는 체계호출이 차단되는 경우(LOCK_NB가 설정되지 않은 경우) 1이고 그렇지 않으면(LOCK_NB가 설정된 경우) 0이다. 이 함수는 다음과 같은 작업을 차례로 수행한다.

ㄱ. 잠그기를 획득해야 하는 경우 새로운 file_lock객체를 얻어서 잠그기연산에 따라 채운다.

ㄴ. filp->f_dentry->d_indoe->i_flock가 가리키는 목록을 탐색한다.

변수의 파일객체와 같은 파일에 대해 FL_LOCK잠그기가 있으며 잠그기해제를 요청한 경우이면 i마디목록과 대역목록에서 file_lock항목을 제거하고 이 잠그기의 대기렬에서 잠들어있는 모든 프로쎄스를 깨우고 file_lock구조체를 해제하고 되돌이한다.

ㄷ. 그렇지 않으면 다시 i마디목록을 탐색하여 존재하는 FL_LOCK중에 요청한것과 충돌하는것이 없는지 확인한다.

i마디목록에는 FL_LOCK쓰기잠그기가 없어야 하고 지금 요청된 작업이 쓰기잠그기인 경우에는 FL_LOCK잠그기가 아무것도 없어야 한다.

그렇지만 프로쎄스는 flock()체계호출을 다시 호출하여 자신이 이미 획득한 잠그기류형을 변경할수 있다.

따라서 핵심부는 언제나 프로쎄스가 같은 파일객체를 참조하는 잠그기를 변경하도록 허잠그기한다.

충돌하는 잠그기를 발견했고 LOCK_NB기발을 1로 지정한 경우 오유코드를 반환한다.

그렇지 않으면 현재프로쎄스를 차단된 프로쎄스의 원형목록에 삽입하고 프로쎄스를 보류한다.

ㄹ. 비호환성(충돌)이 없다면 file_lock구조체를 대역잠그기 목록과 i마디목록에 삽입하고 0(성공)을 반환한다.

④ flock_lock_file()의 반환코드를 반환한다.

4) FL_POSIX잠그기

FL_POSIX잠그기는 언제나 한 프로쎄스와 한 i마디의 조합을 대상으로 한다.

잠그기는 프로쎄스가 죽거나 파일서술자가 닫히면(프로쎄스가 같은 파일을 두번 열었거나 파일서술자를 복제한 경우에도) 자동으로 해제된다.

또한 FL_POSIX잠그기는 절대로 fork()를 통해 자식프로쎄스에 상속하지 못한다.

fcntl()체계호출은 파일잠그기에 사용할 때 변수 세개를 받는다.

열쇠를 걸려는 파일의 파일서술자fd, 잠그기연산을 지정하는 변수 cmd 그리고 flock구조체를 가리키는 지적자fl이다.

Linux 판본 2.6은 flock64구조체를 정의하고있는데 이 구조체는 파일편위와 같이 마당을 위해 64비트마당을 사용한다.

여기서는 flock자료구조위주로 설명하지만 flock64에도 똑같이 적용된다.

FL_POSIX류형의 잠그기는 임의의 파일령역에 열쇠를 걸수 있다.

심지어 1바이트까지도 가능하다.

령역은 flock구조체의 세 마당을 사용하여 지정한다.

l_start는 령역의 시작편위이고 파일의 처음위치(l_whence 마당을 SEEK_SET로 설정한 경우), 현재 파일지적자(l_whence마당을 SEEK_CUR로 설정한 경우), 파일의 끝(l_whence마당을 SEEK_END로 설정한 경우)에서부터 상대적인 값이다.

l_len마당은 파일령역의 길이를 지정한다.(0을 지정하면 령역은 현재 파일의 끝 이후에 대한 모든 쓰기를 포함한다.)

sys_fcntl()봉사루틴은 cmd변수에 설정한 기발값에 따라 다르게 동작한다.

F_GETLK

flock구조체로 서술한 잠그기가 이미 다른 프로쎄스가 획득한 FL_POSIX잠그기와 충돌하는지 검사한다.

충돌하는 경우 flock 구조체를 충돌하는 잠그기의 정보로 채운다.

F_SETLK

flock구조체가 서술하는대로 잠그기를 설정한다. 잠그기를 획득할수 없으면 체계호출은 오유코드를 반환한다.

F_SETLKW

flock구조체가 서술하는대로 잠그기를 설정한다. 잠그기를 획득할수 없으면 체계호출은 차단된다.

즉 호출한 프로쎄스는 잠든다.(sleep 호출)

F_GETLK64, F_SETLK64, F_SETLKW64

flock대신 flock64자료구조를 사용하는것외에는 앞의 설명과 같다.

sys_fcntl()은 잠그기를 획득하기 위해 다음과 같은 과정을 수행한다.

① 사용자공간에서 flock구조체를 읽는다.

② fd에 대응하는 파일객체를 얻는다.

③ 잠그기가 강제적이여야 하는 경우 파일이 공유기억기배치를 가지고있는지 검사한다. 공유기억기배치를 가지고있으면 잠그기생성을 거절하고 EAGAIN오유코드를 반환한다. 이미 다른 프로쎄스가 파일을 사용하고있는 경우이다.

④ 사용자의 flock구조체에 따라 새로운 file_lock구조체를 초기화한다.

⑤ 요청한 잠그기류형이 지정하는 접근방식을 파일이 허가하지 않는 경우 오유코드를 반환하고 완료한다.

⑥ 파일연산에 lock메쏘드가 정의되여있다면 호출한다.

⑦ posix_lock_file()함수를 호출하여 다음 작업을 수행한다.

ㄱ. i마디의 잠그기목록에 있는 각 FL_POSIX잠그기에 대해 posix_locks_conflict()를 호출한다. 함수는 매 잠그기가 요청한 잠그기와 충돌하는지 검사한다. 핵심은 i마디목록안의 같은 령역에 대해 FL_POSIX쓰기잠그기가 없어야 하고 만약 프로쎄스가 쓰기잠그기를 요청하였다면 같은 령역에 대해 FL_POSIX잠그기가 없어야 한다. 그렇지만 같은 프로쎄스가 소유하고있는 잠그기와는 충돌하지 않는다. 이렇게 함으로써 프로쎄스가 자신이 소유한 잠그기의 특성을 바꾸는것을 허용한다.

ㄴ. 충돌하는 잠그기를 발견했을 때 F_SETLK 또는 F_SETLK64 기발을 설정하여 fcntl()을 호출한 경우라면 오유코드를 반환한다. 그렇지 않으면 현재프로쎄스를 보류해야 한다. 이 경우 posix_locks_deadlock()를 호출하여 FL_POSIX잠그기를 기다리는 프로쎄스사이에 교착(deadlock)조전이 발생하는지 검사한다. 그리고 현재프로쎄스를 차단된 프로쎄스들의 원형목록에 삽입하고 보류(suspend)한다.

ㄷ. i마디의 매 목록에 충돌하는 잠그기가 없으면 현재프로쎄스의 모든 FL_POSIX잠그기에 대해 현재프로쎄스가 잠그기를 얻으려는 파일령역과 겹치는 령역이 없는지 검사하고 린접하는 령역을 요청대로 합치거나 쪼갠다. 례를 들어 프로쎄스가 요청한 쓰기잠그기가 이미 읽기열쇠가 걸린 더 넓은 령역에 포함된다면 이전의 읽기잠그기는 서로 떨어져있는 두 령역으로 나뉘고 가운데령역은 새로운 쓰기 잠그기가 보호하게 된다. 겹치는 령역이 발생하면 언제나 새로운 잠그기가 이전것을 대신한다.

ㄹ. 새로운 file_lock구조체를 대역잠그기목록과 i마디목록에 삽입한다.

⑧ 0(성공)을 반환한다.

제 2 절. 파일접근

파일에 접근하는것은 VFS추상계층, 블로크장치, 디스크캐쉬사용 등을 포함하는 복잡한 작업이다.

이 장에서는 이런 기능을 사용하여 핵심부가 파일읽기와 쓰기를 처리하는 방법을 본다.

이 절의 기본주제는 디스크기반의 파일체계에 저장된 정규파일과 블로크장치파일에 똑같이 적용된다.

이제부터 이 두 종류의 파일을 간단히 《파일》이라고 부른다.

이 절에서는 특정한 파일에 대해 읽기 또는 쓰기메쏘드를 호출한 직후부터 본다.

즉 읽기호출에서 요청한 자료를 사용자방식의 프로세스에 전달하는 과정, 쓰기호출에서 요청한 자료를 디스크에 전송하도록 표시하는 과정을 본다.

특히 《파일읽기와 쓰기》에서는 read()와 write()체계호출을 사용해서 정규파일에 접근하는 방법을 설명한다.

프로쎄스가 파일에서 읽기를 수행할 때 자료는 디스크에서 먼저 핵심부주소공간의 완충기로 이동한다.

이 완충기는 캐쉬의 페지에 포함된다. 계속하여 프로쎄스의 사용자주소공간으로 페지를 복사한다.

쓰기연산은 일부 과정이 읽기와 다르지만 기본적으로 반대로 동작한다.

기억기배치에서는 프로쎄스가 직접 정규파일을 자기의 주소공간으로 대응하도록 핵심부에서 허용하는 방법을 설명한다.

이 작업역시 핵심부기억기에 있는 페지를 다루는것을 포함하기때문이다.

마지막으로 직접입출력전송에서는 자신이 직접 캐쉬를 관리하는 응용프로그람을 핵심부에서 어떻게 지원하는하는가를 살펴본다.

1. 파일의 읽기와 쓰기

앞절의 《read()와 write()체계호출》에서 read()와 write()체계호출의 실현방법을 설명하였다.

대응하는 봉사루틴은 결국 파일객체의 read와 write메쏘드를 호출하게 되는데 이것들은 매 파일체계마다 독립적이다.

디스크기반의 파일체계의 경우 이 메쏘드들은 접근할 자료를 포함하는 물리적인 블로크의 위치를 찾아서 자료전송을 시작하도록 블로크장치구동프로그람을 활성화한다.

파일읽기는 페지단위로 진행된다.

핵심부는 언제나 전체 페지자료를 한번에 전송한다. 프로쎄스가 몇바이트를 얻으려고 read()체계호출을 했는데 자료가 RAM에 없으면 핵심부는 새로운 페지틀을 할당하고 이 페지를 파일의 적당한 부분으로 채운 다음 페지캐쉬에 추가하고 마지막으로 요청

한 바이트를 프로쎄스의 주소공간으로 복사한다.

대부분의 파일체계에서 한 페지의 자료를 읽는 과정에서 문제가 되는 부분은 디스크의 어떤 블로크가 요청한 자료를 담고있는지 찾는 일이다.

이 위치를 찾으면 핵심부는 페지입출력연산을 한두번 실행하여 해당 페지를 채운다.

대부분의 파일체계에서 read메쏘드는 generic_file_read()라는 공통함수를 사용하여 실현한다.

디스크기반의 파일에 대한 쓰기연산은 약간 다루기 어렵다.

파일크기가 바뀔수 있으므로 핵심부가 디스크의 물리적블로크를 새로 할당하거나 해제해야 할수도 있다.

물론 이것들을 정확히 어떻게 처리할것인가는 파일체계형에 따라 다르다.

그렇지만 대부분의 디스크기반의 파일체계(례를 들면 Ext2, System V, Coherent, Xenix, Minix 등)에서 write메쏘드는 generic_file_write()공통함수를 사용하여 실현한다.

기록형 또는 망 파일체계인 경우 독자적인 함수를 사용하여 write메쏘드를 실현한다.

1) 파일에서 읽기

대부분의 디스크기반의 파일체계의 정규파일과 모든 블로크장치파일의 read메쏘드를 실현하는 generic_file_read()함수를 보자.

이 함수는 다음과 같은 변수를 받는다.

filp: 파일객체주소

desc: 파일에서 읽은 내용을 보관하기 위한 사용자방식의 기억기령역의 선형주소(linear address)

actor: 읽기동작을 가리키는 구조체

ppos: 읽기를 시작할 파일편위를 담은 변수를 가리키는 지적자(보통 filp파일객체의 f_pos마당)

nonblock: 블로크장치에 대한 읽기인가를 가르킨다.

먼저 함수는 파일객체의 O_DIRECT기발이 설정되여있는가를 검사한다.

설정되여있으면 읽기접근과정에서 페지캐쉬를 무시하고 건너뛴다.

이 경우에 관해서는 《직접입출력전송》에서 설명한다.

여기서는 O_DIRECT기발이 설정되여있지 않다고 가정한다.

함수는 access_ok()를 호출하여 sys_read()체계호출봉사루틴에서 받은 buf와 count 매 변수가 옳은지 검사하고 잘못이 있으면 -DEFAULT오유코드를 반환한다.

모두 정상이면 generic_file_read()는 읽기연산서술자(즉 진행중인 파일읽기연산의 현재상태를 보관하는 read_descriptor_t형태의 자료구조)를 할당한다. 서술자의 마당은

표 2-20과 같다.

표 2-20. 읽기연산서술자마당

형태	마당	설명
size_t	written	전송한 바이트수
size_t	count	전송할 바이트수
char *	buf	사용자방식의 완충기에서 현재 위치
int	error	읽기연산의 오유코드(오유가 없으면 0)

다음으로 do_generic_file_read()함수를 호출한다. 변수에는 파일객체의 지적자 filp, 파일편위지적자 ppos, 방금 할당한 읽기연산서술자의 주소 desc, file_read_actor()함수의 주소를 전달한다. do_generic_file_read()함수는 다음과 같이 동작한다.

① 읽으려는 파일에 해당하는 address_space객체를 얻는다. filp->f_dentry->d_inode->i_mapping에 객체의 주소가 있다.

② 주소공간(address_space)을 소유한 i마디객체를 얻는다. 객체의 주소는 address_space객체의 host마당에 있다. 이 객체는 filp->f_dentry->d_inode가 가리키는 i마디객체와 다른것일수도 있다.

③ 파일의 페지(페지당 4096B)단위자료로 나뉘어져있다고 가정하고 파일지적자 *ppos로부터 요청한 첫번째 바이트를 포함하고있는 페지의 론리적인 index를 구한다. 그리고 페지안에서 요청한 첫번째 바이트의 위치를 offset에 보관한다.

④ 파일지적자가 파일의 《미리 읽기(read-ahead)》창문안에 있는지, 밖에 있는지를 결정한다. 미리읽기에 대해서는 《파일미리읽기》에서 설명한다.

⑤ 요청한 desc->count바이트를 포함하는 모든 페지를 읽는 작업을 매 페지에 대해 반복한다. 매 반복에서 함수는 다음과 같은 단계를 거쳐 한 페지의 자료를 읽는다.

ㄱ. index × 4096 + offset가 i마디객체의 i_size마당에 보관된 크기를 초과하면 반복을 끝내고 ⑥단계로 이동한다.

ㄴ. 요청한 자료를 보관하고있는 페지를 페지캐쉬에서 찾아본다. 앞에서 설명한것처럼 페지캐쉬는 address_space객체의 주소와 파일안에서 페지의 위치(색인)를 사용하여 접근할수 있는 하쉬표이다.

ㄷ. 페지가 페지캐쉬에 없으면 새로운 페지틀을 할당하고 add_page_cache()를 호출하여 이 페지틀을 페지캐쉬에 삽입한다. 페지의 PG_uptodate기발을 지우고 PG_locked기발을 설정한다. 함수는 i단계로 이동한다.

ㄹ. 여기에 도달하면 페지캐쉬에서 페지를 발견한 경우이다. 함수는 페지서술자의 사용계수기를 증가시킨다.

ㅁ. 페지의 PG_uptodate기발을 검사한다. 기발이 설정되여있으면 페지에 보관되여있는 자료는 최신(up-to-date)이다. 함수는 ㅊ단계로 이동한다.

ㅂ. generic_file_readahead()함수를 호출하여 파일에 대한 미리 읽기연산을 활성화할지를 검사한다. 《파일미리읽기》에서 보았지만 이 함수는 페지의 다른 블로크에 대한 입출력자료전송을 시작할수 있다.

ㅅ. 페지의 자료가 유효하지 않으므로 디스크에서 자료를 읽어야 한다. 페지의 PG_locked기발을 설정하여 페지에 대한 배타적인 접근권한을 얻는다. 물론 이전에 시작한 입출력자료전송이 아직 실행중이면 페지에 이미 열쇠가 걸려있을수 있다. 이 경우 페지가 잠그기에서 풀릴 때까지 잠든다. 깨여나면 다시 PG_uptodate기발을 검사하여 다른 자료전송에서 필요한 자료읽기를 실행하였는지를 검사한다. 기발이 1로 설정되여있으면 ㅋ단계로 이동한다. 그렇지 않으면 읽기작업을 계속한다.

ㅇ. 파일의 address_space객체의 readpage메쏘드를 호출한다. 메쏘드에 대응하는 함수는 디스크에서 페지로의 입출력자료전송을 활성화하는 작업을 수행한다. 뒤에서 이 함수가 정규파일과 블로크장치파일에 대해 어떤 작업을 하는지 설명한다.

ㅈ. 페지의 PG_uptodate기발을 검사한다. 입출력자료전송이 아직 끝나지 않았으면 기발은 여전히 0일것이다. 함수는 generic_file_readahead()를 다시 호출하고 입출력자료전송이 완료될 때까지 기다린다.

ㅊ. 이제는 페지에 최신자료가 들어있다. 함수는 generic_file_readahead()함수를 호출하여 파일에 대해 또 다른 미리읽기연산을 활성화할지 검사한다. 《파일미리읽기》에서 보았지만 이 함수는 페지의 다른 블로크에 대한 입출력자료전송을 시작할수 있다.

ㅋ. mark_page_accessed()를 호출하여 PG_referenced기발을 설정한다. 이 기발은 페지가 현재 사용중이며 교환하여 내보내지(swap out)말아야 함을 나타낸다. 사용자가 명시적으로 요청한 페지에 대해서만 이 기발을 설정한다.(즉 미리 읽기가 아닌 경우이다.)

ㅌ. 이제 페지의 자료를 사용자방식의 완충기로 복사해야 한다. 이를 위해 do_generic_file_read()함수는 변수로 그 주소를 전달받은 file_read_actor()함수를 호출한다. file_read_actor()는 다음것들중에서 한가지 단계를 실행한다.

· kmap()를 호출하여 페지가 기억기의 높은 주소에 위치한 경우 영구적인(permanent) 핵심부배치를 설정한다.

· __copy_to_user()를 호출하여 페지자료를 사용자방식의 주소공간으로 복사한다. 이 연산은 프로쎄스를 차단할수도 있다.

· kunmap()를 호출하여 페지의 영구적인 핵심부배치를 모두 해제한다.

· read_descriptor_t서술자의 count, written, buf마당을 갱신한다.

ㅍ. 사용자방식의 완충기에 실제로 전송된 바이트수에 따라 index와 offset국부변수를 갱신한다.

ㅎ. 페지서술자의 사용계수기를 감소시킨다.

ㅠ. read_descriptor_t서술자의 count마당이 0이 아니고 페지에 있는 요청한 바이트를 모두 사용자방식의 주소공간으로 복사하는데 성공했으면 파일에서 다음 페지의 자료에 대해 반복을 계속하기 위해 ⑤-ㄱ단계로 이동한다.

⑥ ppos에 index * 4096 + offset값을 저장한다. 이 값은 이 함수를 다음에 호출했을 때 읽기연산을 실행할 위치이다.

⑦ 파일서술자의 f_reada마당을 1로 설정하여 파일에서 자료를 순차적으로 읽고있다는 사실을 기록한다.

⑧ update_atime()을 호출하여 현재시간을 파일i마디의 i_atime마당에 보관하고 i마디를 《불결한 (dirty)》것으로 표시한다.

2) 정규파일을 위한 readpage메쏘드

앞절에서 본것처럼 do_generic_file_read()는 각 페지를 디스크에서 기억기로 읽기 위해 readpage메쏘드를 반복하여 사용한다.

address_space객체의 readpage메쏘드는 물리적디스크에서 페지캐쉬로 입출력자료전송을 활성화하는 함수의 주소를 보관한다.

정규파일인 경우 보통 이 마당은 mpage_readpages함수를 호출하는 래퍼함수를 가리킨다.

wrapper함수가 필요한 리유는 mpage_readpages함수가 채울 페지서술자page, 그리고 mpage_readpages가 정확한 블로크를 찾는것을 도와주는 get_block함수의 주소 get_block를 변수로 받기때문이다.

이 함수는 파일시작부터의 위치인 블로크번호를 디스크구획에서 상대적인 위치인 론리적인 블로크번호로 변환한다.

두번째 변수는 물론 정규파일이 속한 파일체계의 류형에 따라 다르다.

이 례에서는 ext2_get_block()함수의 주소이다.

mpage_readpages함수는 페지에 포함된 완충기에서부터 페지입출력연산을 시작한다.

필요하면 완충기머리부를 할당하고 앞에서 설명한 get_block메쏘드를 사용하여 디스크에서 완충기를 찾고 자료를 전송한다.

특히 다음과 같은 단계를 실행한다.

① page->buffers마당을 검사한다. NULL이면 create_empty_buffers()를 호출하여 페지에 포함된 모든 완충기에 대해 비동기완충기머리부를 할당한다. 페지의 첫번째 완충기에 대한 완충기머리부의 주소는 page->buffers마당에 보관한다. 매 완충기머리부의 b_this_page마당은 페지에서 다음 완충기의 완충기머리부를 가리킨다.

② 페지안에서 상대적인 파일편위로부터(page->index마당) 페지의 첫번째 블로크의 파일블로크번호를 얻는다.

③ 페지의 매 완충기의 완충기머리부에 대해 다음단계를 실행한다.

ㄱ. BH_Uptodate기발이 설정되여있으면 완충기를 건너뛰고 페지의 다음 완충기에 대해 계속한다.

ㄴ. BH_Mapped기발이 설정되여있지 않으면 get_block라는 변수로 그 주소를 전달받은 파일체계에 독자적인 함수를 호출한다. 이 함수는 파일체계에 있는 디스크에서의 자료구조를 탐색하여 완충기의 론리적인 블로크번호를 찾는다.(론리적블로크번호는 정규파일의 시작부터가 아닌 디스크구획의 시작부터 위치를 나타낸다.)

ㄷ. 파일체계에 독자적인 이 함수는 이 번호를 완충기머리부의 b_blocknr에 보관하고 완충기머리부의 BH_Mapped기발을 설정한다.

드물게 블로크가 정규파일에 속해있으나 응용프로그람이 그 위치에 구멍(hole)을 가지고있어서 블로크를 찾지 못하는 경우가 있다.

이 경우 mpage_readpages는 완충기를 0으로 채우고 완충기머리부의 BH_Uptodate기발을 설정하고 페지의 다음 완충기에 대해 계속한다.

파일체계에 독자적인 함수가 완충기를 갱신하는 블로크입출력연산을 시작했을수도 있으므로 BH_Uptodate기발을 다시 검사한다. BH_Uptodate가 설정되여있으면 페지의 다음 완충기에 대해 계속한다.

ㄹ. 완충기머리부의 주소를 배렬 arr에 보관하고 페지의 다음 완충기에 대해 계속한다.

④ 이제 국부배렬 arr는 최신내용을 포함하지 않은 완충기에 대응하는 완충기머리부의 주소를 담게 된다.

배렬이 비여있으면 페지의 모든 완충기가 유효한 경우이다.

함수는 페지서술자의 PG_uptodate기발을 설정하고 페지를 잠그기에서 풀고 끝낸다.

⑤ 여기까지 하면 arr배렬이 비여있지 않다. mpage_readpages ()가 배렬의 매 완충기머리부에 대해 다음 단계를 실행한다.

ㄱ. BH_Lock기발을 설정한다.

기발이 이미 설정되여있으면 함수는 완충기가 잠그기에서 풀릴 때까지 기다린다.

ㄴ. 완충기머리부의 b_end_io마당을 end_buffer_io_ async함수의 주소로 설정한다.

ㄷ. 완충기머리부의 BH_Async기발을 설정한다.

⑥ arr배렬이 매 완충기머리부에 대해 submit_bh()함수를 호출한다.

이때 연산류형을 READ로 지정한다.

《ll_rw_block()함수》에서 본것처럼 이 함수는 대응하는 블로크의 입출력자료전송을 시작하도록 한다.

3) 블로크장치파일을 위한 readpage메쏘드

블로크장치에 대한 입출력연산은 open()체계호출에서 지정한 블로크장치파일의 i마디가 아니고 서술자의 bd_inode마당이 가리키는 bdev특수파일체계에 속한 블로크장치i마디를 사용한다.(서로 다른 장치파일이 같은 블로크장치를 나타낼수도 있다.)

블로크장치는 대응하는 블로크장치i마디의 i_data마당에 저장된 address_space객체를 사용한다.

정규파일과 달리(정규파일의 address_space객체에 있는 readpage메쏘드는 파일이 속한 파일체계류형에 따라 결정된다.) 블로크장치의 readpage메쏘드는 언제나 같다.

이 메쏘드는 blkdev_readpage()함수로 구현한다.

앞절과 마찬가지로 block_read_full_page()함수를 사용하는데 여기서 두번째 변수는 파일시작위치부터의 블로크번호를 블로크장치시작위치부터의 론리적블로크번호로 변환하는 함수를 가리킨다.

그런데 블로크장치파일인 경우 두 번호는 동일하다. blkdev_get_block함수는 다음 단계를 수행한다.

① 페지의 첫번째 블로크번호가 블로크장치의 크기를 넘지 않았는가를 검사한다.(blk_size[MAJOR(inode->i_rdev)] [MINOR(inode-> i_rdev)]에 보관되여있다.) 넘었다면 오유코드 -EIO를 반환한다.

② 완충기머리부의 b_dev마당을 inode->r_dev로 설정한다.

③ 완충기머리부의 b_blocknr마당을 페지의 첫번째블로크의 파일블로크크번호로 설정한다.

④ 완충기머리부의 BH_Mapped기발을 설정하여 완충기머리부의 b_dev와 b_blocknr마당이 의미있는 값을 보관하고있음을 표시한다.

4) 파일미리읽기

디스크접근은 순차적인 경우가 많다.

정규파일은 디스크에 보관될 때 규모가 큰 린접한 분구들에 보관된다.

그리고 프로그람이 파일을 읽거나 복사할 때 첫번째 바이트로부터 마지막 바이트까지 순차적으로 접근하는 경우가 많다.

따라서 입출력연산 몇번으로도 디스크에서 린접한 분구들을 한꺼번에 많이 가져오게 된다.

《미리읽기(read-ahead)》는 정규파일이나 블로크장치파일의 여러 린접페지의 자료를 실제 요청하기 전에 미리 읽는 기법이다.

미리 읽기를 통해서 디스크조종기는 적은 명령으로 한번에 여러 린접한 분구를 처리하게 되므로 대부분의 경우 읽기는 디스크성능은 물론 체계의 응답시간도 개선해준다.

파일을 순차적으로 읽는 프로세스는 요청한 자료가 대부분 이미 기억기에 있으므로 기다리지 않아도 된다.

그러나 미리읽기는 파일에 대한 《임의접근(random accesss)》인 경우에는 도움이

되지 않는다.

이 경우에는 오히려 페지캐쉬에 불필요한 정보를 보관하게 되므로 공간을 랑비한다고 볼수 있다.

따라서 핵심부는 가장 최근의 입출력접근이 이전것에 대해 순차적이지 않다고 판단하면 미리읽기를 중단한다.

파일의 미리읽기는 몇가지 리유로 복잡한 알고리듬을 요구한다.

- 페지단위로 자료를 읽기때문에 미리 읽기알고리듬은 페지내의 편위를 고려할 필요없이 파일내에 접근하려는 페지위치만 고려하면 된다.

 같은 파일의 페지접근은 접근하는 페지가 서로 가깝게 있기만 하다면 순차적이라고 판단한다.

 《가깝다(close)》는 의미는 후에 정의한다.

- 현재접근이 이전것에 대해 순차적이지 않다면(임의접근) 미리읽기를 처음부터 다시 시작해야 한다.
- 프로쎄스가 같은 페지를 반복해서 접근하는 경우에는 미리읽기를 늦추거나 멈춰야 한다.(파일의 아주 작은 일부분만 사용하는 경우)
- 미리읽기알고리듬은 필요하다면 새로운 페지를 읽어들이도록 저수준입출력장치구동프로그람을 활성화해야 한다.

미리읽기알고리듬은 파일의 련속하는 부분에 대응하는 페지집합을 《미리읽기창문(read-ahead window)》으로 인식한다.

프로쎄스가 요청한 다음 읽기연산이 이 페지집합안에 포함되면 핵심부는 이 파일접근이 이전 접근에 대해 《순차적(sequential)》이라고 판단한다.

미리읽기창문은 프로쎄스가 요청한 페지 또는 핵심부가 미리읽은 페지로 이루어지며 페지캐쉬에 들어있다.

미리읽기창문은 언제나 바로전에 수행한 미리읽기연산이 요청한 페지들을 포함한다.

이것을 《미리읽기그룹》이라고 한다. 프로쎄스가 요청한 다음 연산이 미리읽기그룹에 들어있으면 핵심부는 읽기중인 프로쎄스보다 조금 더 앞서 나가려고 미리읽기창문 다음에 있는 페지를 더 읽을수도 있다.

미리읽기창문이나 미리읽기그룹의 페지가 반드시 최신일 필요는 없다.

디스크에서 전송을 완료하지 않은 경우 이것들을 사용할수 없다.(PG_uptodate기발이 설정되여있지 않다.)

파일객체는 미리읽기와 관련하여 다음과 같은 마당을 포함한다.

f_raend

 미리읽기그룹과 미리읽기창문 다음에 있는 첫번째 바이트위치

f_rawin

 현재 미리읽기창문의 바이트단위길이

f_ralen

현재 미리읽기그룹의 바이트단위길이

f_ramax

다음 미리읽기연산에 대한 문자단위의 최대길이

f_reada

lseek()체계호출을 사용하여 파일지적자를 명시적으로 설정했는지(값이 0), 이전의 read()체계호출을 사용하여 명시적으로 설정했는지(값이 1)를 나타내는 기발

파일을 열 때 이 마당들은 모두 0으로 설정된다. 그림 2-3은 이 마당들을 사용하여 미리읽기창문과 미리읽기그룹을 어떻게 구분하는가를 보여준다.

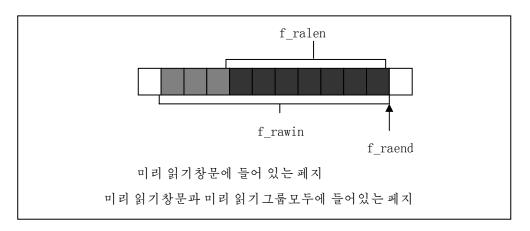



그림 2-3. 미리 읽기창문과 미리읽기그룹

핵심부는 두 종류의 미리읽기연산을 구분한다.

5) 동기적미리읽기연산

읽기가 현재 미리읽기창문밖을 대상으로 할 때 실행된다. 동기적미리읽기연산은 언제나 사용자가 읽기연산에서 요청한 모든 페지와 추가적인 한 페지에 영향을 준다.

연산이 끝나면 미리읽기창문은 미리읽기그룹과 같게 된다.(그림 2-4참고)

6) 비동기적미리읽기연산

읽기가 현재 미리읽기창문안을 대상으로 할 때 실행된다.

비동기적미리읽기연산은 이전의 미리읽기그룹의 페지의 두배를 읽어서 미리읽기창문을 확장하려고 한다.

새로운 미리읽기창문은 이전의 미리읽기그룹과 새 미리읽기그룹으로 구성된다.(그림 2-4참고)

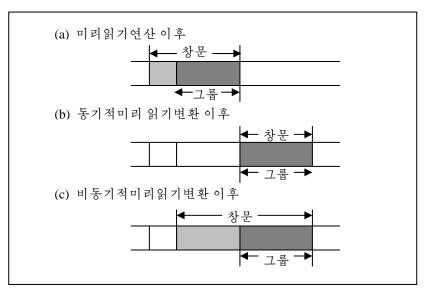



그림 2-4. 미리읽기 그룹과 창문

미리읽기의 동작을 설명하기 위해 사용자가 파일에 대해 read()체계호출을 하였다고 가정하자.

read() 함수는 do_generic_file_read()함수를 호출한다.

do_generic_file_read()함수는 읽으려는 첫번째 페지가 파일의 현재 미리읽기창문에 들어있는지 검사한다.

3가지 경우가 있을수 있다.

- 첫번째 페지가 현재 미리읽기창문밖에 있다. 함수는 파일객체의 f_raend, f_ralen, f_ramax, f_rawin마당을 0으로 설정한다. 그리고 reada_ok변수를 0으로 설정하여 비동기적미리읽기연산을 금지한다.
- 첫번째 페지가 현재 미리읽기창문안에 있다. 이것은 사용자가 파일에 순차적으로 접근하고있음을 의미한다. 함수는 reada_ok변수를 1로 설정하여 비동기적 미리읽기연산을 활성화한다.
- 파일에 한번도 접근한 사실이 없어서 현재 미리읽기창문과 미리읽기그룹이 비여있다.

  또 읽으려는 첫번째페지가 파일의 첫 페지이다. 이 경우 함수는 reada_ok변수를 1로 설정하여 비동기적미리읽기연산을 활성화한다.

do_generic_file_read()함수는 파일객체의 f_ramax마당값을 조정한다.

이 마당은 다음번 미리읽기연산에서 요청할 페지수를 나타낸다.

이 값은 파일에 대한 직전의 미리읽기연산에 의해 결정되지만 do_generic_file_read()는 f_ramax가 언제나 read()체계호출이 요청한 페지수+1보다 큰 값을 저장하게 된다.

그리고 함수는 f_ramax가 언제나 vm_min_readahead 대역변수(보통 3페지)보다 크고 장치별 최대값보다 작은 값을 보관하게 된다.

매 블로크장치는 max_readahead배렬에 값을 정의할수 있다. 이 배렬은 장치의 주번호와 부번호를 통해 참조한다.

장치구동프로그람이 최대값을 지정하지 않으면 핵심부는 vm_max_readahead대역변수의 값(보통 31페지)을 최대값으로 사용한다.

체계관리자는 각각 /proc/sys/vm/min-readahead와 /proc/sys/vm/max-readahead파일에 값을 기록하여 vm_min_readahead와 vm_max-readahead값을 변경할수 있다.

《파일에서 읽기》에서 본것처럼 do_generic_file_read()함수는 최소한 읽기요청에 포함된 매 페지에 대해 한번씩 generic_file_readahead()함수를 여러번 호출한다.

이 함수는 파일과 i마디객체, do_generic_file_read()가 현재 읽으려는 페지의 서술자, read_ok기발값(비동기적미리읽기연산 활성화 또는 비활성화) 등을 변수로 받는다.

페지를 미리읽기위해서 generic_file_readahead()함수는 page_cache_read()를 호출하고 이 함수는 페지를 페지캐쉬에서 탐색하고(삽입할수도 있다.) 대응하는 address_space객체의 readpage메쏘드를 호출하여 입출력자료전송을 요청한다. generic_file_readahead()의 전체적인 동작은 다음과 같다.

기본적으로 이 함수는 동기적인 경우와 비동기적인 경우를 구분한다. 변수로 전달된 페지서술자를 검사해서 서술자의 PG_locked기발이 설정되여있으면 페지는 do_generic_file_read()함수가 시작한 입출력자료전송에 관련한것이므로 미리읽기를 동기적으로 처리한다. 그렇지 않으면 비동기적미리읽기가 가능하다. 다음절에서 PG_locked기발에 따른 동작을 본다.

2. 파일에 쓰기

앞에서 본것처럼 write()체계호출은 자기를 호출한 프로쎄스의 사용자방식의 주소공간에 있는 자료를 핵심부자료구조로 옮긴 다음 다시 디스크로 옮긴다. 파일객체의 write메쏘드는 각 파일체계의 류형이 독자적인 write연산을 정의하도록 한다. 핵심부 2.6에서 매 디스크기반의 파일체계의 write메쏘드는 write연산에 관련된 디스크블로크를 찾아서 자료를 사용자방식의 주소공간에서 페지캐쉬에 속한 페지로 복사하고 페지완충기를 《불결한》(dirty)것으로 표시한다.

Ext2를 포함한 일부 파일체계에서 파일객체의 write메쏘드는 generic_file_write()함수를 사용하여 실현한다. 이 함수는 다음과 같은 변수를 받는다.

file

파일객체지적자

buf

파일에 기록할 문자(character)를 가져올 주소

count

파일에 기록할 문자의 수

ppos

쓰기연산을 시작할 파일편위를 담고있는 변수의 주소

함수는 다음과 같은 연산을 수행한다.

① 변수 count와 buf가 유효한지 검사한다.(이 변수들은 반드시 사용자방식의 주소공간을 가리켜야 한다.) 유효하지 않으면 오유코드 EFAULT를 되돌린다.

② 기록할 파일에 대응하는 i마디객체의 주소 inode를 결정한다.(file-> f_dentry->d_inode->i_mapping->host)

③ 신호기inode->i_sem을 얻는다. 신호기를 사용하기때문에 한번에 한 프로쎄스만 파일에 대한 write()체계호출을 처리할수 있다.

④ file->flags의 O_APPEND기발이 설정되여있으면 정규파일인 경우(블로크장치파일이 아닌 경우) *ppos를 파일의 끝으로 설정하여 모든 새로운 자료를 파일의 끝에 추가하도록 한다.

⑤ 파일의 크기에 대한 몇가지 검사를 실행한다. 레를 들어 쓰기연산은 current->rlim[RLIMIT_SIZE]에 저장된 사용자별한계(3장에 있는 프로쎄스 자원한계)를 넘거나 inode->i_sb->s_maxbytes에 저장된 파일체계한계를 넘도록 정규파일을 확장하면 안된다.

⑥ 현재 시각을 inode->mtime마당(마지막 파일쓰기연산시각)과 inode-> ctime마당(마지막i마디변경시각)에 저장된다. 그리고 i마디객체를 《불결한》(dirty)》것으로 표시한다.

⑦ 파일객체의 O_DIRECT기발을 검사한다. 기발이 설정되있으면 페지캐쉬를 무시한다. 이 경우에 관해서는 이 절의 뒤부분에서 본다. 이 절에서는 O_DIRECT가 설정되여있지 않다고 가정한다.

⑧ 파일의 쓰기연산에 관련된 모든 페지에 대한 반복작업을 시작한다. 매 페지에 대해 다음과 같은 작업을 수행한다.

ㄱ. 페지를 페지캐쉬에서 찾아본다. 페지캐쉬에 없으면 여유페지를 할당하여 새로 페지를 캐쉬에 추가한다.

ㄴ. 페지에 열쇠를 건다. 즉 PG_locked기발을 설정한다.

ㄷ. 예방을 위해 페지소용계수기를 증가시킨다.

ㄹ. kmap()를 호출하여 페지시작선형주소를 얻는다.

ㅁ. i마디의 address_space객체에 있는 prepare_write메쏘드를 호출한다.(file->f_dentry->d_indoe->i_mapping) 메쏘드에 대응하는 함수는 페지에 대한 비동기적완충기머리부를 할당하고 필요하다면 디스크에서 완충기를 읽어온다. 이 함수가 정규파일과 블로크장치파일에 대해 어떻게 동작하는지는 뒤에 나오는 절에서 본다.

ㅂ. __copy_from_user()를 호출하여 사용자방식의 완충기에서 페지로 자료를 복사한다.

ㅅ. i마디의 address_space객체에 있는 commit_write메쏘드를 호출한다.(file->f_dentry->d_indoe->i_mapping) 대응하는 함수는 완충기를 《불결한(dirty)》것으로 표시하여 나중에 디스크에 기록되도록 한다. 이 함수가 정규파일과 블로크장치파일에 대해 어떻게 동작하는지는 뒤에서 본다.

ㅇ. kunmap()를 호출하여 ⑧-d단계에서 설정한 높은자리기억기배치를 해제한다.

ㅈ. 페지의 PG_referenced기발을 설정한다. 이것은 기억기회수알고리듬에서 사용한다.

ㅊ. PG_locked기발을 지우고 페지의 열쇠가 풀리기를 기다리는 프로쎄스를 깨운다.

ㅋ. ⑧-c에서 증가시킨것을 취소하려고 페지사용계수기를 감소시킨다.

⑨ 이제 파일의 쓰기연산에 관련된 모든 페지를 처리하였다. 마지막으로 기록한 문자다음을 가리키도록 **ppos의 값을 변경한다.

⑩ 파일의 O_SYNC기발을 검사한다. 기발이 설정되여있으면 generic_osync_inode()를 호출하여 핵심부가 페지의 모든 《불결한》 완충기를 디스크에 청소하게 하고 입출력자료전송이 완료될 때까지 현재프로쎄스를 중단하다.

⑪ inode->i_sem신호기를 해제한다.

⑫ 파일에 기록한 문자수를 반환한다.

1) 정규파일에 대한 prepare_write와 commit_write메쏘드

address_space객체의 prepare_write와 commit_write메쏘드는 generic_file_write()가 실현하는 일반적인 쓰기연산을 정규파일과 블로크장치파일에 대해 특수화시킨다. 두 메쏘드는 쓰기연산에서 각 페지에 대해 호출한다.

매 디스크기반의 파일체계는 자신의 prepare_write메쏘드를 정의한다. 이 메쏘드는 read연산과 마찬가지로 공통함수에 대한 래퍼함수이다. 례를 들어 Ext2 파일체계의 prepare_write메쏘드는 다음과 같은 함수로 정의된다.

ext2_get_block()함수는 앞에서 본 《파일에서 읽기》에서 설명하였다. 이 함수는 파일의 상대적인 블로크번호를 물리적인 블로크장치에서 자료를 나타내는 론리적인 블로크번호로 변환한다.

block_prepare_write()함수는 다음 단계를 수행하여 파일에 있는 페지에 대해 완충기와 완충기머리부를 준비한다.

① page->buffers마당을 검사하여 NULL이면 create_empty_buffers()를 호출하여 페지의 모든 완충기에 대해 완충기머리부를 할당한다. 페지에서 첫번째 완충기의 완충기머리부주소는 page->buffers마당에 저장한다. 매 완충기머리부의 b_this_page마당은 페지에서 다음 완충기의 완충기머리부를 가리킨다.

② 페지에 포함된 쓰기연산의 대상이 되는 완충기에 대응하는 완충기머리부에 대해

다음을 수행한다.

ㄱ. 함수는 BH_Mapped기발이 설정되여있지 않으면 다음 단계를 수행한다.

· 함수의 주소를 변수로 받은 파일체계에 따르는 함수를 호출한다. 함수는 파일체계에 있는 디스크에서의 자료구조에서 완충기의 론리적블로크번호(정규파일의 시작부터의 번호가 아닌 디스크구획의 시작부터의 번호)를 찾는다. 이 파일체계에 따르는 함수는 이 번호를 대응하는 완충기머리부의 b-blocknr마당에 저장하고 BH_Mapped기발을 설정한다. 이 파일체계에 따르는 함수는 파일에 대한 새로운 물리적인 완충기를 할당할수도 있다. 이 경우 BH_New기발을 설정한다.

· BH_New기발의 값을 검사한다. 기발이 설정되여있으면 unmap_underlying_metadata()를 호출하여 완충기캐쉬가 디스크의 해당 블로크를 참조하는 《불결한》 완충기를 소유하고있지 않게 한다. 또한 쓰기연산이 전체 완충기를 재기록하지 않으면 함수는 완충기를 0으로 채운다. 그리고 페지의 다음 완충기를 처리하도록 한다.

ㄴ. 쓰기연산이 전체 완충기를 재기록하지 않고 완충기의 BH_Uptodate기발이 설정되여있지 않다면 함수는 블로크에 대해 ll_rw_block()를 호출하여 디스크에서 내용을 읽어온다.

③ ②-ㄴ단계에서 시작한 모든 읽기연산이 완료될 때까지 현재프로쎄스를 중지한다.

prepare_write메쏘드에서 되돌아가면 generic_file_write()함수는 페지를 사용자방식의 주소공간의 자료로 갱신한다. 그리고 address_space객체의 commit_write메쏘드를 호출한다. 이 메쏘드는 대부분의 디스크기반파일체계에서 generic_commit_write()함수로 구현한다.

generic_commit_write()함수는 다음 단계를 수행한다.

① block_commit_write()함수를 호출한다. 이 함수는 쓰기연산의 영향을 받는 페지의 모든 완충기를 대상으로 매 완충기에 대해 BH_Uptodate와 BH_Dirty 기발을 설정하고 완충기머리부를 BHF_DIRTY목록과 i마디의 《불결한》완충기목록에 삽입한다.(이미 목록애 있지 않는 경우) 함수는 balance_dirty()함수를 호출하여 체계전체의 《불결한》완충기수가 제한된 값을 넘지 않도록 한다.

② 쓰기연산이 파일을 증가시켰는가를 검사한다. 그렇다면 함수는 파일의 i마디의 i_size마당을 갱신하고 i마디객체를 《불결한》것으로 표시한다.

2) 블로크장치파일에 대한 prepare_write와 commit_write메쏘드

블로크장치파일에 대한 쓰기연산은 정규파일에 대한 쓰기연산과 아주 비슷하다. 사실 블로크장치파일의 address_space객체에 있는 prepare_write메쏘드는 보통 다음과 같이 실현된다.

이 함수는 단지 앞에서 설명한 block_prepare_write()함수를 호출한다. 차이점은 두번째의 변수에 있다. 이 변수는 파일시작부터의 위치를 나타내는 파일블로크번호

를 블로크장치시작부터의 위치를 나타내는 론리적인 블로크번호로 변환하는 함수이다. 블로크장치파일인 경우 이 두 수자는 같다.

블로크장치파일에 대한 commit_write메쏘드는 간단한 래퍼함수로 실현된다.

블로크장치파일에 대한 commit_write메쏘드는 정규파일에 대한 commit_write메쏘드와 같다. 차이점은 이 메쏘드는 쓰기연산이 파일을 증가시켰는가를 검사하지 않는다는 점이다. 마지막 위치에 문자를 덧붙인다고 해서 블로크장치파일을 증가시킬수는 없다.

3. 기억기배치

기억기령역은 디스크기반의 파일체계의 정규파일 또는 블로크장치파일의 일부분에 대응할수 있다. 즉 기억기령역에 있는 페지의 한 바이트에 대한 접근을 핵심부가 파일의 해당 바이트에 대한 연산으로 변환한다. 이 기법을 《기억기배치 (memory mapping)》라고 부른다.

기억기배치에는 두가지 종류가 있다.

1) 공유 (shared)

기억기령역의 페지에 대해 쓰기연산을 하게 되면 디스크의 파일을 변경한다. 그리고 어떤 프로쎄스가 공유기억기에 배치된 페지에 쓰기를 하는 경우 이 파일을 기억기배치하는 다른 모든 프로쎄스가 그 변화를 알수 있다.

2) 비공개 (private)

프로쎄스가 파일에 대해 쓰기를 하지 않고 읽기만을 위해서 배치를 생성하는 경우에 사용한다. 읽기용으로 사용하는 경우 비공개배치는 공유배치보다 더 효률적이다. 그러나 비공개배치페지에 대해 쓰기요청을 하면 이 페지가 더는 파일의 페지와 배치되지 않는다.

쓰기는 디스크의 파일내용을 변경하지 못하며 같은 파일에 접근하는 다른 프로쎄스는 이 변화를 알지 못한다.

프로쎄스는 새로운 기억기배치를 생성하기 위해서 mmap()체계호출을 사용한다. 프로그람작성자는 체계호출의 변수로 MAP_SHARED기발이나 MAP_PRIVATE기발을 지정해야 한다. 추측한것처럼 전자는 공유배치에 사용하고 후자는 비공개배치에 사용한다. 배치를 생성하고 나면 프로쎄스는 새로운 기억기령역의 기억기위치에서 파일에 저장된 자료를 읽을수 있다. 공유기억기배치인 경우 프로쎄스는 같은 기억기위치에쓰기를 함으로써 대응하는 파일을 변경할수 있다. 프로쎄스는 기억기배치를 제거하거나 줄이기 위해서 munmap()체계호출을 사용한다.

일반적을 공유기억기배치인 경우 대응하는 기억기령역은 VM_SHARED기발을 1로 설정하고 비공개인 경우에는 VM_SHARED기발을 0으로 설정한다. 후에 보지만 읽기전

용인 공유기억기배치인 경우 이 규칙에서 벗어날 때도 있다.

3) 기억기배치자료구조

다음과 같은 자료구조의 조합으로 기억기배치를 나타낸다.

- 배치된 파일에 대응하는 i마디객체
- 배치된 파일의 address_space객체
- 이 파일에 대해 여러 프로쎄스가 실행한 대 배치에 대한 파일객체
- 파일에서 서로 다른 각 배치에 대한 vm_area_struct서술자
- 파일을 배치하는 기억기령역에 할당된 매 페지들에 대한 페지서술자

매 i마디객체의 i_mmapping마당은 파일의 address_space객체를 가리킨다. 매 address_space객체의 i_mmap와 i_mmap_shared마당은 파일을 배치하는 모든 기억기령역을 포함하고있는 2중련결목록의 첫번째 요소를 가리킨다. 두 마당이 모두 NULL인 경우에 이 파일은 어떤 기억기령역에도 배치되여있지 않다. 이 목록은 기억기령역을 나타내는 vm_area_struct서술자를 포함하고있으며 vm_next_share와 vm_pprev_share마당이 목록을 실현한다.매 기억기령역서술자의 vm_file마당은 배치된 파일에 대한 파일객체주소를 담는다. 이 마당이 NULL이면 이 기억기령역은 기억기배치에 사용되지 않는다. 파일객체는 핵심부가 기억기령역을 소유하고있는 프로쎄스와 배치된 파일을 알수 있게 해주는 마당을 포함한다.

배치된 시작위치는 기억기령역서술자의 vm_pgoff마당에 보관된다. 이 마당은 페지크기단위로 파일에서의 편위를 나타낸다. 배치된 파일부분의 길이는 기억기령역의 길이와 같으므로 vm_start와 vm_end 마당으로부터 구할수 있다.

공유기억기배치된 페지는 언제나 페지캐쉬에 들어있다. 비공개기억기배치된 페지는 변경되지 않는 동안에는 페지캐쉬에 들어있다. 프로쎄스가 비공개기억기배치페지를 변경하려면 핵심부는 이 페지들을 복제하고 프로쎄스페지표에서 원본페지들을 대신하여 복제프레임으로 대체한다. 이것은 앞장에서 설명한 《쓰기복사》기법을 응용한 실례이다. 원본페지들은 여전히 페지캐쉬에 있지만 복제한것으로 대체되였기때문에 더는 기억기배치에 속하지 않는다. 그리고 복제된 프레임은 디스크의 파일을 나타내는 자료를 포함하지 않으므로 페지캐쉬에 삽입되지 않는다.

핵심부는 서로 다른 매 파일체계에 대해 독자적인 기억기배치기구를 위한 후크(hook)를 제공한다. 기억기배치실현의 핵심은 파일객체의 mmap메쏘드이다. 대부분의 디스크기반의 파일체계와 블로크장치파일의 경우 이 메쏘드는 generic_file_mmap()이라는 함수를 통해 실현하며 다음 부분에서 설명한다.

파일-기억기배치는 요구페지화에서 설명한 요구페지화(demand paging) 기구에 의존한다. 새로 설정된 기억기배치는 빈 기억기령역으로 아직 페지를 포함하지 않는다.

프로쎄스가 령역내부의 주소를 참조함에 따라 페지오유(Page Fault)가 발생한다. 페지오유처리기는 기억기령역에 대해 nopage메쏘드가 정의되여있는가를 검사한다. 정의되여있지 않으면 기억기령역은 디스크의 파일에 대한 배치가 아니다. 정의되여있다면 메쏘드는 블로크장치에 접근하여 페지를 읽도록 한다. 대부분의 디스크기반의 파일체계와 블로크장치파일의 nopage메쏘드는 filemap_nopage()함수를 통해 실현한다.

4) 기억기배치생성

프로쎄스는 새로운 기억기배치를 생성하기 위해서 다음과 같은 변수를 전달하여 mmap()체계호출함수를 호출한다.

- 배치될 파일을 나타내는 파일서술자
- 배치될 파일부분의 첫번째문자를 나타내는 파일안의 편위
- 배치될 파일부분의 길이
- 기발집합. 프로쎄스는 요청한 기억기배치의 종류를 지정하기 위해 MAP_SHARED기발이나 MAP_PRIVATE기발을 명시적으로 설정해야 한다.
- 기억기령역에 대한 하나이상의 접근권한을 나타내는 집합. 읽기접근(PROT_READ), 쓰기접근(PROT_WRITE), 실행접근(PROT_EXEC) 권한이 있다.
- 선형주소(선택항목이다) 핵심부는 새로운 기억기령역이 시작할 위치를 지정하는 암시로서 이 주소를 사용한다. MAP_FIXED기발을 지정하고 핵심부가 지정된 선형주소에서 시작하는 새로운 기억기령역을 할당하지 못할 경우 체계호출은 실패한다.

mmap()체계호출은 새로운 기억기령역과 시작위치의 선형주소를 변환한다. 호환성문제때문에 80x86구조에서 핵심부는 mmap()를 위해 체계호출표에서 두개의 입구점(entry)을 사용한다 두 입구점의 색인값은 90과 192이다. 전자는 old_mmap()봉사루틴에 해당하며(이전의 C서고에서 사용) 후자는 sys_mmap2봉사루틴에 해당한다.(최근의 C서고에서 사용한다.) 두 봉사루틴의 차이점은 체계호출의 변수 6개의 전달방법뿐이다. 둘 다 do_mmap_pgoff()함수를 호출한다. 이제 파일을 배치하는 기억기령역을 생성할 때 수행하는 단계를 설명한다.

① 배치할 파일에 대해 mmap파일연산이 정의되여있는지를 검사한다. 정의되여있지 않다면 오유코드를 반환한다. 파일연산표의 mmap가 NULL값이면 대응하는 파일을 배치할수 없음을 나타낸다.(레를 들면 등록부의 경우)

② 파일객체의 get_unmapped_area메쏘드가 정의되여있는지를 검사하여 정의되여있으면 호출한다. 그렇지 않으면 arch_get_unmapped_area()함수를 호출한다. 80x86구조에서 독자적인 메쏘드는 프레임완충기계층에서만 사용하므로 더 설명하지 않는다. arch_get_unmapped_area()는 새로운 기억기령역에 대해 선형

주소공간을 할당한다.

③ 일반적인 일관성검사외에 요청한 기억기배치의 종류와 파일을 열 때 지정한 기발을 비교한다. 체계호출의 파라메터로 받은 기발은 요청한 배치종류를 나타내고 파일객체의 f_mode값은 파일을 열었을 때 지정한 기발을 나타낸다. 이 두 종류의 정보에 대해 다음과 같은 검사를 실행한다.

ㄱ. 쓰기가능한 공유기억기배치를 요청한 경우 쓰기용으로 파일을 열었으며 추가방식(open()체계호출의 O_APPEND기발)으로 열지 않았다는 사실을 확인한다.

ㄴ. 공유기억기배치를 요청한 경우 파일에 강제적인 잠그기(mandatory_lock)가 없음을 확인한다.

ㄷ. 모든 기억기배치에 대해 읽기를 허용하여 파일을 열었는지 확인한다.

이중에서 만족하지 않는 사항이 하나라도 있으면 오유코드를 반환한다.

④ 새로운 기억기령역서술자의 vm_flags마당값을 초기화할 때 파일의 접근권한과 요청한 기억기배치종류에 따라 VM_READ, VM_WRTIE, VM_EXEC, VM_SHARED, VM_MAYREAD, VM_MAYWRITE, VM_MAYEXEC, VM_MAYSHARE기발을 설정한다. 성능을 높이기 위해서 쓰기금지된 공유기억기배치에 대해서는 VM_SHARED기발의 설정을 해제한다. 기억기령역이 페지에 대해 프로쎄스의 쓰기가 금지되였으므로 비공개배치와 똑같이 다룰수 있기때문이다. 그러나 실제로는 파일을 공유하는 다른 프로쎄스가 이 기억기령역의 페지에 접근하는것을 허용한다.

⑤ 기억기령역서술자의 vm_file마당을 파일객체의 주소로 초기화하고 파일의 사용계수기를 증가시킨다.

⑥ 배치된 파일에 대해 객체의 주소와 기억기령역서술자의 주소를 변수로 전달하여 mmap메쏘드를 호출한다. 대부분의 파일체계에서 이 메쏘드는 generic_file_mmap()함수로 실현하며 다음과 같은 연산을 수행한다.

ㄱ. 쓰기가능한 공유기억기배치를 요청한 경우 파일의 address_space객체의 writepage메쏘드가 정의되여있는가를 검사한다. 정의되여있지 않으면 EINVAL오유코드를 반환한다.

ㄴ. 파일의 address_space객체의 readpage메쏘드가 정의되여있는가를 검사한다. 정의되여있지 않으면 ENOEXEC오유코드를 반환한다.

ㄷ. 현재시각을 파일의 i마디의 i_atime마당에 저장하고 i마디를 《불결한》것으로 표시한다.

ㄹ. 기억기령역서설자의 vm_ops마당을 generic_file_vm_ops표의 주소로 초기화한다. 이 표의 nopage메쏘드이외의 모든 메쏘드는 NULL이다.

nopage메쏘드는 filemap_nopage()함수로 실현한다.

⑦ vma_link()는 기억기배치가 비공개인지, 공유인지 여부에 따라서 기억기령역서술자를 address_space객체의 i_mmap목록 또는 i_mmap_shared목록에 삽입한다.

5) 기억기배치제거

프로쎄스가 기억기배치를 제거하려면 다음과 같은 변수를 전달하여 munmap()체계호출함수를 호출한다.

- 제거할 선형주소구간의 시작위치주소
- 제거할 선형주소구간의 길이

munmap()체계호출은 각 종류의 기억기령역을 제거하거나 크기를 줄이는데 사용할수 있다. 실제로 이 체계호출의 sys_munmap()봉사루틴은 do_munmap()함수를 호출한다. 그러나 파일을 배치한 기억기령역의 경우 해제할 선형주소구간에 포함된 각 기억기령역에 대해 다음과 같은 단계를 추가로 실행한다.

① remove_shared_vm_strucet()를 호출하여 기억기령역서술자를 address_space객체목록(i_mmap또는 i_mmap_shared)에서 제거한다.

② unmap_fixup()함수를 실행하면서 한 기억기령역전체를 제거하면 파일사용계수기를 감소시키고 새로운 기억기령역을 생성하면 파일사용계수기를 증가시킨다. 즉 배치를 제거하는것이 령역안에 구멍(hole)을 생성하는 경우이다. 령역이 줄어드는 경우라면 파일사용계수기에는 변화가 없다.

쓰기가능한 공유기억기배치를 제거하는 경우에 여기에 포함된 페지의 내용을 디스크에서 청소할 필요는 없다. 이 페지들은 계속 페지캐쉬에 남아있으므로 계속 디스크캐쉬로 동작한다.

6) 기억기배치에 대한 요구페지요구화

효률을 높이기 위해서 기억기배치가 생성되면 즉시 페지들이 할당되지 않으며 가능한 마지막순간까지 즉 프로쎄스가 페지중하나를 참조하여 《페지오유》례외가 발생할 때까지 할당을 낮춘다.

핵심부는 오유가 발생한 주소에 대응하는 페지표입구점을 검사하고 입구점이 비여있으면 do_no_page()함수를 호출한다.

do_no_page()함수는 페지틀할당과 페지표를 갱신과 같이 모든 요구페지화에 공통적인 연산을 수행한다. 또한 관련기억기령역이 nopage메쏘드를 정의하고있는가를 검사한다. 여기서는 이 메쏘드가 정의되여있는 경우 함수가 수행하는 작업을 설명한다.

① nopage메쏘드를 호출한다. 이 메쏘드는 요청한 페지를 포함하는 페지틀

의 주소를 반환한다.

② 기억기배치가 비공개인 경우 프로쎄스가 페지에 쓰기를 시도하면 방금 읽은 페지의 복사본을 만들고 이것을 비활성(inactive)페지목록에 삽입하여 더는 《쓰기복사》오유가 발생하지 않게 한다. 이후 단계에서 함수는 nopage메쏘드가 반환한 페지대신 새로운 페지를 사용하여 사용자방식의 프로쎄스가 nopage메쏘드가 반환한 페지를 변경할수 없게 한다.

③ 새로운 페지틀이 프로쎄스에 할당되였음을 나타내기 위해서 프로쎄스는 기억기서술자의 rss마당을 증가시킨다.

④ 오유가 발생한 주소에 대응하는 페지표입구점을 페지틀의 주소와 기억기령역의 vm_page_prot마당에 포함된 페지접근권한으로 설정한다.

⑤ 프로쎄스가 페지에 쓰기를 시도하면 페지표입구점의 Read/Write와 Dirty비트를 1로 설정한다. 이 경우 페지틀을 프로쎄스에 배타적으로 할당하거나 페지를 공유한다. 두 경우 모두 페지에 쓰기를 허용해야 한다.

요구페지화알고리듬의 핵심은 기억기령역의 nopage메쏘드이다. 간단히 말해서 이 메쏘드는 프로쎄스가 접근하는 페지를 포함하고있는 페지틀의 주소를 반환해야 한다. 메쏘드의 구현은 페지를 포함하고있는 기억기령역의 종류에 따라 다르다.

파일을 디스크로 배치하는 기억기령역을 다루면서 먼저 nopage메쏘드는 요청한 페지가 페지캐쉬에 있는지 찾아본다. 페지를 찾지 못하면 메쏘드는 페지를 디스크에서 읽어야 한다. 대부분의 파일체계는 filemap_nopage()함수를 사용하여 nopage메쏘드를 구현한다.

이 함수는 다음과 같은 세개의 변수를 받는다.

area

요청한 페지를 포함하고있는 기억기령역의 서술자주소이다.

address

요청한 페지선형주소이다.

unused

filemap_nopage()함수가 사용하지 않는 nopage메쏘드의 변수이다.

filemap_nopage()함수는 다음과 같은 단계를 실행한다.

① area->vm_file마당에서 파일객체의 주소를 얻는다. file->f_dentry-> d_inode->i_mapping에서 address_space객체의 주소를 얻는다. addresss_space객체의 host마당에서 i마당객체의 주소를 얻는다.

② area의 vm_start와 vm_offset마당을 사용하여 address에서 시작하는 페지에 대응하는 자료파일안에서의 편위를 결정한다.

③ 파일편위가 파일크기를 초과하는가를 검사한다. 초과하는 경우 NULL을 반환한

다. 이것은 새로운 페지할당에서 실패하였다는것을 의미한다. 다만 오유추적기가 ptrace()체계호출을 사용하여 다른 프로쎄스를 추적하다가 페지오유가 발생한 경우는 례외인데 이 경우에 관해서는 설명하지 않는다.

④ find_page()를 호출하여 address_space객체와 같은 파일편위로 지정하는 페지를 페지캐쉬에서 찾는다.

⑤ 페지가 페지캐쉬에 없으면 기억기령역의 VM_RAND_READ기발값을 검사한다. madvise()체계호출을 사용하여 이 기발값을 변경할수 있다. 기발이 설정되여있으면 사용자응용프로그람이 해당 파일에 대해 지금 접근하는 페지외에 페지를 더 읽지 않을것임을 나타낸다.

・VM_RAND_READ기발이 설정되여있으면 page_cache_read()를 호출하여 요청한 페지만을 디스크에서 읽는다.

・VM_RAND_READ기발이 설정되여있지 않으면 page_cache_read()를 여러번 호출하여 요청한 페지뿐만아니라 기억기령역안에 있는 린접한 페지들의 분구를 읽도록 한다. 분구의 길이는 page_request변수에 저장되여있다. 기본값은 3페지인데 체계관리자가 /proc/sys/vm/page-cluster특수파일에 값을 써서 바꿀수 있다.

다음으로 함수는 4단계로 돌아가서 페지캐쉬탐색을 계속한다.(page_cache_read()함수가 실행되는 동안 프로쎄스는 차단되여있어야만 한다.)여기에 이르면 페지가 페지캐쉬에 들어있다. 페지의 PG_uptodate기발을 검사한다. 기발이 설정되여있지 않으면(페지의 내용이 최신이 아님) 다음 단계를 실행한다.

ㄱ. PG_locked기발을 설정하여 페지에 열쇠를 건다. 필요하다면 잠든다.

ㄴ. address_space객체의 readpage메쏘드를 호출하여 입출력자료전송이 시작하게 된다.

ㄷ. wait_on_page()를 호출하여 입출력자료전송이 완료될 때까지 기다린다.

⑥ 이제 페지는 최신상태로 되였다. 함수는 기억기령역의 VM_SEQ_READ기발을 검사한다. madvise()체계호출을 사용하여 이 기발의 값을 변경할수 있다. 기발이 설정되여있으면 사용자응용프로그람이 배치된 파일의 페지를 순차적으로 참조할것임을 나타낸다. 따라서 적극적으로 페지를 미리 읽고 접근한 다음에는 즉시 해제해야 한다. 기발이 설정되여있으면 nopage_sequential_readahead(0를 호출한다. 이 함수는 크기가 고정된 큰 미리읽기창문을 사용한다. 그 길이는 대략 하부블로크장치의 최대 미리읽기창문의 크기이다. 기억기령역서술자의 vm_raend마당에 현재 미리읽기창문의 마지막위치가 들어있다. 함수는 요청한 페지가 현재 미리읽기창문의 중간에 이르면(대응하는 페지를 먼저 읽음으로써) 미리읽기창문을 앞으로 옮긴다. 그리고 함수는 기억기령역의 페지구역에서 요청한 페지보다 뒤쪽에 있는 페지를 해제해야 한다. 함수가 기억기령역의 n번째 미리 읽기창문을 읽었으면 (n-3)번째 창문에 속한 페지를 디스크로 청소한다.

⑦ mark_page_accessed()를 호출하여 요청한 페지에 접근했음을 표시한다.

⑧ 요청한 페지의 주소를 반환한다.

7) 불결한 기억기배치페지를 디스크로 흘리기

프로쎄스는 msync()체계호출을 사용하여 공유기억기배치에 속하는 불결한 페지를 디스크로 쓰기한다.

이 체계호출은 선형주소구간의 시작주소, 구간의 길이 그리고 다음과 같은 의미를 지닌 기발집합을 변수로 받는다.

MS_SYNC

입출력연산을 끝낼 때까지 프로쎄스를 연기하도록 체계호출에 요청한다. 이렇게 함으로써 호출하는 프로쎄스는 체계호출이 끝나면 기억기배치의 모든 페지를 디스크에 청소하였다는 사실을 확인할수 있다.

MS_ASYNC

호출하는 프로쎄스를 연기하지 않고 즉시 반환하도록 체계호출에 요청한다.

MS_INVALIDATE

기억기배치에 포함된 모든 페지를 프로쎄스주소공간에서 제거하도록 체계호출에 요청한다.(실제로는 구현되지 않았다.)

sys_msync()봉사루틴은 선형주소구간에 포함된 각 기억기령역에 대해 msync_interval()을 호출한다. msync_interval()함수는 다음과 같은 연산을 수행한다.

① 기억기령역서술자의 vm_file마당이 NULL이거나 VM_SHARED기발이 0이면 0을 되돌린다.(기억기령역은 파일의 쓰기가능한 공유기억기배치가 아니다.)

② filemap_sync()함수를 호출한다. 이 함수는 기억기령역에 포함된 선형주소구간에 대응하는 페지표입구점을 탐색한다. 발견한 각 페지에 대해서 flush_tlb_page()를 호출하여 대응하는 TLB(translation lookaside buffer)를 청소하고 페지를 불결한것으로 표시한다.

③ MS_SYNC기발이 0이면 끝낸다. 그렇지 않으면 다음 단계를 계속 수행하여 기억기령역의 페지를 디스크로 청소하고 모든 입출력자료전송이 완료될 때까지 기다린다. 현재 핵심부의 마지막안정판에서는 MS_INVALIDATE기발을 무시한다.

④ 파일의 i마디의 i_sem신호기를 얻는다.

⑤ filemap_fdatasync()함수를 호출한다. 이 함수는 파일의 address_space객체의 주소를 변수로 받는다. 함수는 address_space객체의 불결한 페지목록에 속한 모든 페지에 대해서 다음과 같은 단계를 수행한다.

ㄱ. 페지를 불결한 페지목록에서 열쇠가 걸려있는 페지목록으로 옮긴다.

ㄴ. PG_Dirty기발이 설정되여있지 않으면 목록의 다음 페지에 대해서 계속한다.(이 페지를 이미 다른 프로쎄스가 청소하였다.)

ㄷ. 페지의 사용계수기를 증가시키고 사용계수기에 열쇠를 건다. 필요하면 기다린다.

ㄹ. 페지의 PG_Dirty기발을 지운다.

ㅁ. 페지의 address_space객체의 writepage메쏘드를 호출한다.

ㅂ. 페지의 사용계수기의 열쇠를 해제한다.

블로크장치파일과 대부분의 디스크기반파일체계의 writepage메쏘드는 block_write_full_page()함수를 호출하는 래퍼함수이다. 이 메쏘드는 파일시작위치부터의 위치를 나타내는 블로크번호를 디스크구획에서의 블로크위치를 나타내는 론리적인 블로크번호로 변환하는 파일체계마다 독자적인 함수를 block_write_full_page()에 전달하는데 사용한다. Block_write_full_page()함수도 앞서 설명한 block_read_full_page()함수와 아주 비슷하다. 이 함수는 페지에 대해 비동기완충기머리부를 할당하고 매 완충기머리부에 대해 WRITE연산을 지정하여 submit_bh()함수를 호출한다.

⑥ 파일객체에 fsync메쏘드가 정의되여있는가를 검사한다. 정의되여있다면 실행한다. 정규파일의 경우 이 메쏘드는 보통 파일의 i마디객체를 디스크에 청소하는것으로 끝난다. 그러나 블로크장치파일의 경우는 sync_buffers()를 호출하고 장치의 모든 불결한 완충기에 대한 입출력자료전송을 활성화한다.

⑦ filemap_fdatawait()함수를 실행한다. 함수는 address_space객체의 열쇠가 걸린 페지목록의 매 페지에 대해서 페지의 열쇠가 해제될 때까지 즉 페지에 대해 진행중인 입출력자료전송이 완료될 때까지 기다린다.

⑧ 파일의 i_sem신호기를 해제한다.

4. 직접입출력전송

지금까지 본것처럼 Linux 2.6에서는 파일체계를 통해 정규파일에 접근하는 기법과 하부블로크장치파일의 블로크를 참조하여 파일에 접근하는 기법, 파일기억기배치를 설정하여 파일에 접근하는 기법 등에 근본적인 차이는 없다고 할수 있다. 그러나 아주 복잡한 프로그람(직접 캐쉬를 관리하는 응용프로그람)의 경우 전체 입출력자료전송기구를 직접 조종하려고 할수 있다. 고성능자료기지봉사기의 경우를 례를 들어 보면 자료기지봉사기는 자료기지에 대한 질문의 특성에 따라서 자신의 캐쉬기구를 실현한다. 이런 종류의 프로그람에는 핵심부의 페지캐쉬가 도움이 되지 않으며 다음과 같은 리유로 하여 오히려 방해가 된다.

- 이미 기억기(사용자준위의 디스크캐쉬)에 있는 디스크자료와 같은 자료를 저장하기 위해 많은 페지들을 랑비한다.
- 페지캐쉬와 미리 읽기를 처리하는 불필요한 명령때문에 read() 와 write()체계호출이 느려진다. 파일기억기배치에 관련된 페지화연산의 경우에도 마찬가지이다.
- read(), write()체계호출에서는 디스크에서 사용자기억기로 자료를 직접 전송하지 않고 디스크에서 핵심부완충기로 그리고 핵심부완충기에서 사용자기억기로

전송이 두번 일어난다.

블로크하드웨어장치는 새치기와 직접기억기접근(DMA, Direct Memory Access)을 통해서 처리해야 하며 이것은 핵심부방식에서만 가능하므로 직접 캐쉬기능을 처리하는 응용프로그람을 실현하려면 어느 정도의 핵심부자원이 필요하다.

Linux판본이 2.6에서는 페지캐쉬를 건너뛰는 《직접입출력전송》이라는 간단한 기법을 제공한다. 핵심부는 각 직접입출력전송에 대해서 디스크조종기를 프로그람화하여 자료를 직접 캐쉬능을 처리하는 응용프로그람의 사용자방식의 주소공간과 주고받도록 한다.

앞에서 설명한것처럼 자료전송은 비동기적으로 진행된다. 자료전송중에 핵심부가 현재프로세스를 전환할수 있으며(다른 프로쎄스로 순서짜기할수 있다.) CPU가 사용자방식으로 돌아왔을 때 자료전송을 시작한 프로쎄스가 교체되여 나가게 될수 있다. 일반적인 입출력자료전송의 경우 디스크캐쉬페지를 사용하므로 이 방식이 잘 동작한다.

교체되여 나가지 않는 핵심부가 디스크캐쉬를 소요하고있으며 핵심부방식의 모든 프로쎄스가 디스크캐쉬를 볼수 있다. 그러나 직접입출력전송은 프로쎄스의 사용자방식의 주소공간에 속한 페지의 자료를 대상으로 한다. 핵심부는 이 페지를 핵심부방식에 있는 모든 프로쎄스가 볼수 있게 하고 자료전송진행중에 이 페지가 교체되여 나가지 않도록 해야 한다. 이를 위해 《직접접근완충기(direct access buffer)》를 사용한다.

직접접근완충기는 직접입출력자료전송을 위해 예약된 물리적인 페지들의 집합으로 구성된다. 이 완충기는 직접 고속완충응용프로그람의 사용자방식의 페지표와 핵심부페지표(각 프로쎄스의 핵심부방식의 페지표) 모두에 배치되여있다. 각 직접접근완충기는 kiobuf자료구조로 되여있으며 이 자료구조의 마당은 표 2-21과 같다.

표 2-21. 직접접근완충기 서술자마당

형	마 당	설 명
int	nr_pages	직접접근완충기에 들어있는 페지수
int	array_len	map_array마당의 여유요소수
int	offset	직접접근완충기의 첫번째 페지에서 의미있는 자료의 편위
int	length	직접접근완충기에 있는 의미있는 자료의 길이
struct page **	maplist	직접접근완충기에 있는 페지들을 가리키는 페지서술자지시자목록 (일반적으로 map_array마당을 가리킨다.)
unsigned int	locked	직접접근완충기에 있는 모든 페지의 잠그기 기발

struct page *[]	map_array	페지서술자지시자 129개의 배렬
struct buffer_ head *	bh	미리 할당된 완충기머리부지시자 1024개의 배렬
unsigned long []	blocks	론리적블로크번호 1024개의 배렬
atomic_t	io_count	입출력이 진행중인지 나타내는 원자적인 (atomic)기발
int	errno	마지막 입출력연산의 오유번호
void(*) (struct kiobuf *)	end_io	완료메쏘드(completion method)
wait_queue headt	wait_qreue	입출력이 완료되길 기다리는 프로쎄스의 대기렬

직접고속완충응용프로그람이 파일에 직접 접근하여 한다고 가정하자. 먼저 응용프로그람은 O_DIRECT기발을 설정하여 파일을 연다. open()체계호출의 봉사과정에서 dentry_open()함수는 이 기발값을 검사한다. 기발이 설정되여있으면 함수는 alloc_kiovec()를 호출하여 새로운 직접접근완충기서술자를 할당하고 그 주소를 파일객체의 f_iobuf마당에 저장한다. 완충기는 처음에 페지들을 포함하지 않으며 따라서 서술자의 mr_pages마당값은 0이다. 그러나 alloc_kiovec()는 완충기머리부 1024개를 미리 할당하며 완충기머리부주소는 서술자의 bh배렬에 저장한다. 이 완충기머리부로 하여 직접고속완충응용프로그람이 파일에 직접 접근하면서 차단되는 경우가 발생하지 않는다. 이 접근방법의 부족점은 한번에 전송될수 있는 자료전송의 단위가 최대 512kB라는 점이다.

다음으로 직접 고속완충응용프로그람이 O_DIRECT로 열었던 파일에 대해 read() 또는 write()체계호출을 하였다고 가정하자. 이 절의 앞에서 언급한것처럼 generic_file_read()와 generic_file_wirte()함수는 기발의 값을 검사하여 기발이 설정되여있으면 별도의 경우로 처리한다. 례를 들어 generic_file_read()함수는 다음과 같은 부분코드를 실행한다.

함수는 파일지적자, 파일크기, 요청한 문자수 등의 현재값을 검사하고 generic_file_direct_IO를 호출한다. 변수로는 READ연산류형, 파일객체지적자, 사용자방식의 완충기주소, 요청한 바이트수, 파일지적자를 전달한다. generic_file_write()함수도 마찬가지다. 다만 generic_file_direct_IO함수에 WRITE연산류형을 전달한다.

generic_file_direct_IO()함수는 다음과 같은 단계를 수행한다.

① 파일객체의 f_iobuf_lock열쇠를 검사하고 설정한다. 이미 열쇠가 설정되여있다면 f_iobuf에 저장된 직접접근완충기서술자를 다른 직접입출력전송에서 사용하고있는것

이므로 함수는 새로운 직접접근완충기서술자를 림시로 할당하여 이후 단계에서 사용한다.

② 파일지적자편위와 요청한 문자수가 파일의 블로크크기의 배수인가를 검사한다. 배수가 아니면 EINVAL을 반환한다.

③ 파일의 address_space객체의 direct_IO메쏘드(filp->f_dentry-> d_inode ->i_mapping)가 정의되여있는지 검사한다. 정의되여있지 않으면 EINVAL을 반환한다.

④ 직접고속완충응용프로그람이 파일에 직접 접근하더라도 체계의 다른 응용프로그람이 페지캐쉬를 통해 파일에 접근할수 있다. 자료손실을 막으려고 직접입출력전송을 시작하기 전에 디스크영상과 페지캐쉬를 동기화한다. 함수는 filemap_fdatasync()함수를 호출하여 파일의 기억기배치에 속한 불결한 페지를 디스크로 청소한다.

⑤ fsync_inode_data_buffers()함수를 호출하여 write()체계호출에 의해 갱신된 불결한 페지를 디스크에 청소하고 입출력자료전송이 완료될 때까지 기다린다.

⑥ 4단계에서 시작한 입출력연산이 완료될 때까지 기다리기 위해 filemap_fdatawait()함수를 호출한다.

⑦ 순환을 시작하며 전송할 자료를 512kB단위의 뭉치(chunk)로 나눈다. 함수는 각 뭉치에 대해 다음과 같은 단계를 수행한다.

ㄱ. 직접접근완충기와 뭉치에 대응하는 사용자준위완충기사이의 배치를 설정하기 위해 map_user_kiobuf()를 호출한다. 함수는 배치설정을 위해 다음을 수행한다.

· expand_kiobuf()를 호출하여 직접접근완충기서술자에 들어있는 배렬이 너무 작으면 새로운 페지서술자주소의 배렬을 할당한다. 그러나 여기서는 map_array마당의 입구점129개가 512kB뭉치를 배치하는데 충분하므로 문제되지 않는다.(추가적인 페지가 필요한 경우는 완충기가 페지맞춤(align)되지 않은 경우이다.)

· 뭉치안에 있는 모든 사용자페지에 접근하여 (페지오유가 발생하는 경우에는 페지를 할당한다.) 페지주소를 직접접근완충기서술자의 maplist마당이 가리키는 배렬에 저장한다.

· nr_pages, offset, length마당을 적절히 초기화하고 locked마당을 0으로 설정한다.

ㄴ. 파일의 address_space객체에 있는 direct_IO메쏘드를 호출한다.

ㄷ. 연산류형이 READ이면 mark_dirty_kiobuf()를 호출하여 직접접근완충기가 배치한 페지를 불결한것으로 표시한다.

ㄹ. unmap_kiobuf()를 호출하여 뭉치와 직접접근완충기사이의 배치를 해제하고 다음 뭉치에 대해 작업을 계속한다.

⑧ 함수가 1단계에서 림시직접접근완충기를 할당했으면 이것을 해제한다. 그렇지 않으면 파일객체의 f_iobuf_lock열쇠를 해제한다.

거의 모든 경우 direct_IO메쏘드는 generic_direct_IO()함수에 대한 래퍼이며 블로

크장치에서 물리적블로크위치를 계산하는 파일체계마다 독자적인 함수의 주소를 generic_direct_IO()함수에 변수로 전달한다. generic_direct_IO함수는 다음 단계를 수행한다.

① 현재 뭉치에 대응하는 파일부분의 각 블로크에 대해 파일체계마다 독자적인 함수를 호출하여 론리적인 블로크번호를 구하고 이 번호를 직접접근완충기서술자의 blocks배렬에 잇는 입구점하나에 저장한다. Linux의 최소블로크크기가 512B이므로 이 배렬의 입구점수 1024개로 충분하다.

② brw_kiovec()함수를 호출한다. 이 함수는 blocks배렬의 각 블로크에 대해 직접접근완충기서술자의 bh배렬에 저장된 완충기머리부를 사용하여 submit_bh()함수를 호출한다. 직접 입출력연산은 완충기입출력 또는 페지입출력연산과 비슷하다. 그러나 완충기머리부의 b_end_io메쏘드는 end_buffer_io_syne()나 end_buffer_io_async()가 아닌 end_buffer_io_kiobuf()라는 함수로 설정된다. 이 메쏘드는 kiobuf자료구조의 마당을 처리한다. brw_kiovec()는 입출력자료전송이 완료되여야만 되돌이한다.

제 3 절. Ext2, Ext3 파일체계

이 절에서는 특정파일체계와 호상작용하기 위해 핵심부가 해야 하는 일을 본다. Ext2(2차확장파일체계)는 Linux고유의 파일체계이고 대부분의 Linux체계에서 사용하고있으므로 가장 적절한 파일체계이다.또한 Ext2는 최신의 파일체계특성을 고성능으로 제공한다. 물론 다른 조작체계를 위해 설계된 다른 파일체계들도 새롭고 흥미있는 요구사항을 실현하고있겠지만 여기서는 여러 파일체계의 차이점에 대해서는 보지 않는다.

Ext2의 일반적인 특징에서는 Ext2를 소개하고 필요한 자료구조를 설명한다. 파일체계가 디스크에 자료를 저장하기 위한것이므로 두 종류의 자료구조를 다루어야 한다. Ext2디스크자료구조에서는 Ext2가 디스크에 저장한 자료구조를 살펴보고 Ext2기억기자료구조에서는 디스크에 있는 자료를 기억기에 복제했을 때의 자료구조를 본다.

그 다음에는 파일체계에서 실행되는 연산을 설명한다. Ext2파일체계생성에서는 디스크구획에 Ext2파일체계를 생성하는 방법을 설명한다. 다음으로 디스크를 사용할 때 핵심부가 항상 실행하는 작업을 설명한다. 이중 대부분은 i마디와 자료블로크를 저장하기 위해 디스크공간을 할당하는 비교적 저수준인 작업이다.

마지막으로 Ext2가 발전한 다음 단계인 Ext3파일체계를 간단히 본다.

1. Ext2의 일반적인 특징

Unix계렬의 조작체계에서는 서로 다른 여러 종류의 파일체계를 사용한다. 모든 파

일체계의 파일은 stat()와 같은 POSIX API가 요구하는 공통속성을 포함하고있지만 각 파일체계의 실현은 서로 다르다.

최초의 Linux판본은 Minix파일체계를 기반으로 했지만 Linux가 성장하면서 확장파일체계(Ext FS)를 도입하였다. 이 파일체계는 몇가지 중요한 확장을 포함했지만 성능이 만족스럽지 못하였다. 그후 1994년에 2차확장파일체계(Ext2)를 도입하였다. Ext2는 몇가지 새로운 특성은 물론 매우 효률적이고 안전하기때문에 지금 가장 널리 사용하는 Linux파일체계로 되였다.

Ext2의 효률성은 다음과 같은 특성때문이다.

- Ext2파일체계를 생성할 때 체계관리자는 예상하는 평균파일크기에 따라 최적의 블로크크기를(1024-4096B) 선택할수 있다. 례를 들어 평균파일크기가 수kB이하일 때 내부단편화(internal fragmentation)를 줄일수 있으므로 1024블로크크기가 좋다. 내부단편화는 파일의 길이와 파일을 저장하는 디스크공간의 크기차이때문에 발생한다. 반면에 파일크기가 수kB이상인 경우에는 디스크전송회수를 줄여 체계부하를 줄일수 있으므로 큰 블로크크기가 좋다.
- Ext2파일체계를 생성할 때 체계관리자는 예상하는 파일수에 따라 같은 크기의 구획에 들어갈 i마디수를 선택할수 있다. 이렇게 함으로써 실제 사용할수 있는 디스크공간을 최대화할수 있다.
- 파일체계는 디스크블로크 몇개를 그룹으로 관리한다. 각 그룹에는 서로 린접하는 자리길에 저장된 자료블로크i마디를 포함한다. 이 구조로 하여 단일 블로크그룹내에 저장된 파일에 접근하는데 필요한 디스크탐색시간은 평균보다 낮아진다.
- 파일체계는 디스크자료블로크를 사용하기 전에 정규파일에 미리 할당한다. 따라서 파일크기가 증가할 때 몇개의 블로크가 이미 물리적으로 린접한 위치에 예약되여있기때문에 파일단편화를 줄일수 있다.
- 빠른 기호련결을 지원한다. 기호련결의 경로명이 60B이하이면 i마디에 경로명을 저장한다. 따라서 자료블로크를 읽지 않고 경로명변환이 가능하다.
- 이 외에도Ext2에는 파일체계를 안전하고 융통성있게 해주는 다른 특징이 있다.
- 체계파피(crash)로 인한 영향을 최소화하고 파일갱신전략을 실현하였다. 례를 들어 파일에 대해 새로운 하드련결을 생성하면 디스크i마디에 있는 하드련결계수기수가 먼저 증가하고 적절한 등록부에 새로운 이름을 추가한다. 이렇게 함으로써 i마디를 변경한 다음 등록부변경이전에 하드웨어고장이 발생하더라도 등록부의 일관성을 유지할수 있다. 물론 i마디의 하드련결계수기는 잘못되여있다. 이 파일을 삭제하더라도 심각한 결과를 일으키지는 않으며 단지 파일의 자료블로크를 자동으로 회수할수 없을뿐이다. 갱신순서가 반대로 이루어지는 경우에는 (등록부를 먼저 변경하고 i마디를 갱신하는 경우) 같은 하드웨어고장으로 인해 위험한 불일치(inconsistency)가 발생할수 있다. 원본 하드련결을 삭제하면

자료블로크도 디스크에서 제거할수 있는데 새로운 등록부항목은 더는 존재하지 않는 i마디를 가리킬것이다. 후에 이 i마디번호를 다른 파일에 사용하게 되면 이전 등록부항목에 쓰는 작업이 새로운 파일을 손상시킬수 있다.

- 기동시에 파일체계상태에 대한 일관성검사를 자동으로 제공한다. 이 검사는 e2fsck라는 외부프로그람이 수행한다. 이 프로그람은 체계충돌이후 지정된 수만큼 파일체계를 탑재한 후(각 탑재연산후에 계수기가 증가한다.) 또는 가장 최근에 수행한 검사이후 지정된 시간이 지났을 때 검사를 시작한다.
- 변경불가능파일(파일을 수정, 삭제하거나 이름을 변경할수 없다.), 추가가능 파일(파일의 맨끝에 자료를 덧붙이는것만 가능하다.)을 지원한다.
- 새로운 파일의 그룹ID에 Unix체계V Release(SVR4), BSD실현과 호환성을 제공한다. SVR4에서 새로운 파일은 자신을 생성한 프로쎄스의 그룹식별자를 가진다. BSD에서 새로운 파일은 자신을 포함하는 웃준위등록부의 그룹식별자를 물려받는다. Ext2에는 둘중 어떤방식을 사용할지를 지정하는 탑재선택항목이 있다.

Ext2파일체계는 안정적이고 완성도가 높은 프로그람이며 최근에는 큰 변화가 없었다. 그렇지만 몇가지 새로운 기능추가가 론의되고있다. 그중 어떤것은 이미 외부패치형태로 코드가 작성되여있기도 하고 계획만 있는 경우도 있으며 실현을 위한 마당이 Ext2 i마디에 추가되여있는 경우도 있다. 론의되고있는 주요기능으로 다음과 같은것이 있다.

블로크단편화

응용프로그람이 자주 큰 파일을 다루기때문에 체계관리자는 디스크접근을 위해 일반적으로 큰 블로크크기를 선택한다. 결과적으로 작은 파일이 큰 블로크에 저장되여 많은 디스크공간을 랑비한다. 여러 파일을 같은 블로크에 속한 서로 다른 단편에 저장할수 있게 하여 이 문제를 해결할수 있다.

접근조종목록

파일의 사용자를 세 부류(소유자, 그룹, 기타)로 분류하는 대신 각 파일에 특정사용자 또는 사용자집합에 대한 접근권한을 나타내는 접근조종목록(ACI, Access Control List)을 련결한다.

압축된 파일과 암호화된 파일의 처리

이 항목들은 새로운 파일을 생성할 때에만 지정할수 있다. 파일을 사용하면 자동으로 파일을 압축하거나 암호화하여 디스크에 저장하도록 한다.

론리적삭제

삭체취소항목을 사용하여 사용자가 요청하면 이전에 삭제한 파일내용을 쉽게 복구할수 있다.

기록형기능

기록형기능을 사용하면 파일체계가 비정상적으로 탑재해제되였을 때(례를 들면 체계

가 중단되였을 때) 자동으로 실행되며 시간이 많이 걸리는 파일체계검사가 필요없게 된다.

Ext2는 대부분의 Linux배포판회사에서 가장 신임하는 파일체계이고 Ext2만큼 철저하게 검사해서 사용하는 파일체계는 없다.

2. Ext2디스크자료구조

모든 Ext2구획의 첫번째 블로크는 Ext2파일체계가 관리하지 않는다. 이 블로크는 구획기동분구를 위해 예약되여있다. Ext2구획의 나머지 부분은 블로크그룹 단위로 나눈다. 각 블로크그룹의 내부배치는 그림 2-5와 같다. 그림에서 알수 있는것처럼 어떤 자료구조는 반드시 한 블로크에 저장해야 하고 어떤것은 한 블로크이상을 사용한다. 파일체계의 모든 블로크그룹은 같은 크기이며 순차적으로 저장된다. 그러므로 핵심부는 블로크그룹의 옹근수색인값으로부터 디스크에서 블로크그룹이 차지한 위치를 구할수 있다.

핵심부는 가능하면 한 파일의 자료블로크를 같은 블로크그룹에 저장하려 하므로 블로크그룹은 파일단편화를 줄이는 효과가 있다. 블로크그룹의 매 블로크는 다음의것들 가운데서 한가지 정보를 포함한다.

- 파일체계의 초블로크 복사본
- 블로크그룹서술자그룹의 복사본

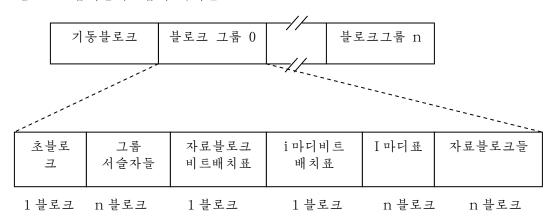



그림 2-5. Ext2 구획과 Ext2 블로크그룹의 내부배치

- 자료블로크비트배치표
- i마디의 그룹
- i마디의 비트배치표
- 파일에 속한 자료의 덩어리 즉 한 자료블로크

어떤 블로크가 의미있는 정보를 전혀 포함하지 않으면 여유(free)블로크라고 한다.

그림에서 볼수 있는것처럼 초블로크와 그룹서술자는 각 블로크그룹에 복제되여있다. 핵심부는 블로크그룹0에 속한 초블로크와 그룹서술자만 사용하며 나머지 초블로크와 그

룹서술자는 변경되지 않은채 남아있다. 사실 핵심부는 이것들을 읽어보지도 않는다. /sbin/e2fsck프로그람이 파일체계상태의 일관성검사를 수행할 때 블로크그룹 0에 저장한 초블로크와 그룹서술자를 읽어서 다른 모든 블로크그룹에 복사한다. 자료가 파괴되여 블로크그룹 0의 초블로크 또는 그룹서술자를 사용할수 없을 때 체계관리자는 /sbin/e2fsck에 첫번째 블로크그룹외의 블로크그룹에 저장한 초블로크와 그룹서술자를 사용하도록 지시할수 있다. 일반적으로 중복된 이 복사본들은 /sbin/e2fsck가 Ext2구획을 일관성있는 상태로 되돌리는데 충분한 정보를 담고있다.

구획에는 블로크그룹이 몇개나 있는가? 이것은 구획의 크기와 블로크크기에 따라 결정된다. 한가지 제약조건은 그룹내에서 사용중인 블로크와 여유블로크를 구분하기 위해 사용하는 블로크비트배치표를 반드시 단일블로크에 저장해야 한다는 점이다. 따라서 블로크크기를 B단위로 나타낸 값을 b라고 할 때 각 블로크그룹에는 최대 8*b개 블로크가 있을수 있다. 따라서 구획의 블로크수를 s라고 할 때 블로크그룹의 수는 대략 s/(8*b)이다.

례를 들어 블로크크기가 4kB인 Ext2구획 8GB가 있다고 하자. 이 경우 각 4kB블로크비트배치표는 자료블로크 32K개를 나타낼수 있으므로 128MB가 된다. 따라서 최대 64블로크그룹이 필요하다. 블로크크기가 작을수록 블로크그룹개수가 늘어나게 된다.

1) 초블로크

Ext2디스크초블로크는 ext2_super_block구조체에 저장된다.

이 구조체의 마당은 표 2-22에 있다. __u8, __u16, __u32자료형태는 각각 길이가 8, 16, 32bit인 부호없는 (unsigned)수를 나타내고 __s8, __s16, __s32자료형태는 각각 8, 16, 32bit인 부호있는 (signed)수를 나타낸다.

표 2-22. Ext2초블로크마당

형	마 당	설 명
__u32	s_inodes_count	총 i마디수
__u32	s_blocks_count	파일체계 크기(블로크 단위)
__u32	s__r_blocks_count	예약된 블로크수
__u32	s_free_blocks_count	여유 블로크수
__u32	s_free_inodes_count	여유 i마디수
__u32	s_free_data_block	사용가능한 첫번째 블로크 번호(언제나 1)
__u32	s_log_block_size	블로크크기
__u32	s_log_frag_size	단편크기

__u32	s_blocks_per_group	그룹당 블로크수
__u32	s_frags_per_group	그룹당 단편수
__u32	s_inodes_per_group	그룹당 i마디수
__u32	s_mtime	마지막 탑재연산시간
__u32	s_wtime	마지막 쓰기연산시간
__u16	s_mnt_count	탑재연산수
__u16	s_max_mnt_count	검사전 탑재 연산수
__u16	s_magic	식별번호(magic number)
__u16	s_state	상태기발
__u16	s_crrors	오유를 검출했을 때 동작
__u16	s _minor_rev_level	세부개정수준
__u32	s_lastcheck	마지막 검사시간
__u32	s_checkinterval	검사사이의 시간간격
__u32	s_creator_os	파일체계가 생성된 os
__u32	s_rev_level	개정준위
__u16	s_def_resuid	예약된 블로크의 기본 UID
__u16	s_def_resgid	예약된 블로크 의 기본 GID
__u32	s_first_ino	예약되지 않은 첫번째 i마디번호
__u16	s_inode_size	디스크우의 i마디구조 크기
__u16	s_block_group_nr	이 초블로크의 블로크그룹
__u32	s_feature_compat	호환가능한 특징비트배치표
__u32	s_feature_incompat	호환되지 않는 특징비트배치표
__u32	s_feature_ro_compat	읽기전용으로 호환되는 특징비트배치표
__u8[16]	s_uuid	128bit 파일체계식별자
char[16]	s_bolumi_name	볼륨이름
char[64]	s_last_mounted	마지막 탑재지점의 경로이름
__u32	s_algorithm_usage_bitmap	압축을 위해 사용됨
__u8	s_prealloc_blocks	미리 할당하는 블로크수
__u8	s_prealloc_dir_blocks	등록부를 위해 미리 할당하는 블로크수
__u16	s_padding1	문자경계에 맞춤
__u32[204]	s_reserved	1024B를 맞추기 위한 빈 공간

s_inodes_count마당은 Ext2파일체계에 있는 i마디수를 저장하고 s_block_count 마당은 블로크수를 저장한다.

s_

log_block_size마당은 블로크크기를 2의 지수로 나타내는데 단위는 1024B이다. 따라서 0은 1024B블로크를 나타내고 1은 2048B블로크를 나타낸다. s_log_frag_size마당은 현재 s_log_block_size와 같다.(블로크단편화가 아직 실현되지 않았기때문이다.)

s_blocks_per_group, s_frags_per_group, s_inodes_per_group마당은 각각 블로크그룹의 블로크수, 단편수, i마디수를 나타낸다.

어느 정도의 디스크블로크는 초사용자(또는 s_def_resuid와 s_def_resgid 마당을 통해 지정한 사용자 또는 사용자그룹)를 위해 예약되여있다. 이 블로크는 일반사용자가 사용할수 있는 여유블로크가 더는 없을 경우에도 체계관리자가 파일체계를 사용할수 있게 한다.

s_mnt_count, s_max_mnt_count, s_lastcheck,s_checkinterval마당은 기동과정에서 Ext2파일체계를 자동으로 검사하도록 한다. 이 마당들은 탑재연산을 지정된 수만큼 실행했거나 마지막 일관성검사이후 지정된 시간이 경과했을 때 /sbin/e2fsck를 실행하게 한다.(두 종류의 검사를 동시에 사용할수 있다.) 파일체계가 탑재해제되지 않았거나(례를 들면 체계붕괴이후) 핵심부가 파일체계에서 어떤 오유를 발견했을 때도 기동시간에 일관성검사를 수행한다. s_state마당은 파일체계를 탑재한 상태거나 제대로 탑재해제되지 않았을 때 0을 저장하고 제대로 탑재해제되였을 때 1을 저장하며 오유를 포함하고있을 때 2를 저장한다.

2) 그룹서술자와 비트배치표

각 블로크그룹에는 자기만의 그룹서술자가 있다.

표 2-23에서 ext2_group_desc구조체마당을 보여준다.

표 2-23　　　　Ext2그룹서술자마당

형	마　당	설　명
__u32	bg_block_bitmap	블로크비트배치표의 블로크번호
__u32	bg_inode_bitmap	i마디비트배치표의 블로크번호
__u32	bg_inode_table	첫번째 i마디표블로크의 블로크번호
__u16	bg_free_blocks_count	그룹안에 있는 여유블로크수
__u16	bg_free_inodes_count	그룹안에 있는 등록부수
__u16	bg_used_dirs_count	그룹안에 있는 등록부수

__u16	bg_pad	문자정렬을 위한 빈 공간
__32[3]	bg_reserved	24B를 채우기 위한 빈 공간

새로운 i마디와 자료블로크를 할당할 때 bg_free_blocks_count, bg_free_inodes_count, bg_used_dirs_count마당을 사용한다. 이 마당은 매 자료구조를 할당하는데 가장 적절한 블로크를 선택할 때 사용한다. 비트배치표는 비트의 련속으로서 값이 0인 경우는 대응하는 i마디 또는 자료블로크가 여유있다는 의미고 값이 1인 경우는 사용중이라는 의미이다. 각 비트배치표는 반드시 단일블로크에 저장해야 하며 블로크크기는 1024, 2048, 4096B일수 있으므로 비트배치표하나는 블로크 8192, 16384, 32768개의 상태를 나타낼수 있다.

3) i마디표

i마디표는 린접하는 련속된 블로크로 이루어져있으며 각 블로크는 미리 정의된 수의 i마디를 포함한다. i마디표에서 첫번째 블로크의 블로크번호는 그룹서술자의 bg_inode_table마당에 저장된다.

모든 i마디의 크기는 128B이다. 1024B블로크는 i마디 8개를 포함하고 4096B블로크는 i마디 32개를 포함한다. i마디표가 얼마나 많은 블로크를 차지하는지 알려면 그룹에 속하는 전체 i마디수(초블로크의 s_inodes_per_group마당에 저장되여있다.)를 블로크당 i마디수로 나누면 된다.

매 Ext2 i마디는 ext2_inode구조체로 이루어져있다.

표 2-24에서 이 구조체마당을 볼수 있다.

표 2-24. Ext2디스크 i마디마당

형	마 당	설 명
__u16	i_mode	파일류형과 접근권한
__u16	i_uid	소유자식별자
__u32	i_size	파일길이(B단위)
__u32	i_atime	마지막 파일접근시간
__u32	i_ctime	i마디가 마지막으로 변경된 시간
__u16	i_mtime	파일의 내용이 마지막으로변경된 시간
__u32	i_dtime	파일삭제시간
__u16	i_gid	그룹식별자
__u16	i_links_count	하드련결수
__u32	i_blocks	파일의 자료블로크수

__u32	i_frags	파일기발
union	i_osd1	특정조작체계
__u32[EXT2_N_BLOCKS]	i_block	자료블로크에 대한 지시자
__u32	i_generation	파일판본(NFS용)
__u32	i_file_acl	파일접근조종목록
__u32	i_dir_acl	등록부접근조종목록
__u32	i_faddr	단편주소
union	osd2	특정조작체계정보

POSIX명세와 관련한 많은 마당은 VFS의 i마디객체의 대응하는 마당과 류사하다. 나머지는 Ext2의 특정실현과 관련있으며 주로 블로크할당을 처리한다.

특히 i_size마당은 파일의 실제길이를 B단위로 나타내는 반면에 i_blocks마당은 파일에 할당된 자료블로크수(512B단위)를 나타낸다.

i_size와 i_blocks값이 항상 서로 관련된것은 아니다. 언제나 정수개의 블로크에 파일을 저장하기때문에 비여있지 않은 파일은 최소한 자료블로크 하나를 포함한다.(단편화가 아직 실현되지 않았기때문에) i_size는 512 * i_blocks보다 작을수도 있다. 반면에 뒤에 나오는 《파일구멍》에서 보지만 파일에 구멍(hole)이 있을수도 있다. 이 경우에 i_size는 512 * i_blocks보다 클수도 있다.

i_block마당은 파일에 할당된 자료블로크를 식별하는데 사용하는 블로크를 가리키는 지적자 EXT2_N_BLOCKS개(보통 15)의 배렬이다.

i_size마당을 위해 32bit가 예약되여있으므로 파일크기는 4GB로 제한된다. 실제로 i_size마당의 제일 웃준위비트는 사용하지 않으므로 최대파일크기는 2GB로 제한되다. 그러나 Ext2파일체계는 HP의 Alpha와 같은 64bit 방식에서 더 큰 파일을 허용하도록 《불결한 책략(dirty trick)》을 도입하였다. 즉 정규파일에서 사용되지 않는 i마디의 i_dir_acl마당을 i_size마당의 32bit확장으로 사용하는것이다. 따라서 파일크기를 i마당에 64bit로 정수로 저장할수 있다. Ext2파일체계의 64bit판본은 32bit판본과 어느 정도 호환한다고 할수 있다. 64bit방식에서 생성한 Ext2파일체계를 32bit방식에서 탑재할수 있으며 그 반대도 가능하기때문이다. 그러나 32bit방식에서는 O_LARGEFILE기발을 설정하지 않으면 큰 파일에 접근할수 없다.

VFS모형에서는 각 파일의 i마디번호가 서로 달라야 한다. Ext2에서는 파일의 i마디번호를 디스크에 저장할 필요가 없다. 블로크그룹번호와 i마디표의 상대적인 위치로부터 i마디번호를 구할수 있기때문이다. 례를 들어 각 블로크그룹에 i마디가 4096개 있고 디스크에서 i마디 13021의 주소를 원한다고 하자. i마디는 세번째 블로크그룹에 속해있

으며 디스크주소는 대응하는 i마디표의 733번째입구점에 저장되여있다. Ext2의 루틴은 디스크에서 적절한 i마디서술자를 빨리 얻기 위해 i마디번호를 열쇠(key)로 사용한다.

4) 여러 파일류형의 디스크블로크사용법

Ext2가 인식하는 여러 류형의 파일은(정규파일, 관, 기타) 자료블로크를 서로 다른 방법으로 사용한다. 어떤 파일은 자료를 저장하지 않으며 따라서 자료블로크가 전혀 필요없다. 여기서는 표 2-25에서 각 류형의 보관사항을 본다.

표 2-25. Ext2파일류형

file_type	설 명
0	알수 없음
1	정규파일
2	등록부
3	문자장치
4	블로크장치
5	이름붙은(named)관
6	소케트
7	기호련결

정규파일

정규파일은 가장 일반적인 경우이며 이 장의 대부분이 정규파일에 관한 내용이다. 정규파일에 자료를 저장하자면 자료블로크가 있어야 한다. 처음 정규파일을 생성할 때에는 파일이 비여있으므로 자료블로크가 없어도 된다. 또한 truncate()나 open()체계호출을 사용하여 다시 빈파일로 만들수 있다. 이것들 모두는 실제로 자주 발생하는 경우이다. 례를 들어 >filename문자렬이 들어있는 쉘명령을 실행하면 쉘은 빈 파일을 생성하거나 존재하는 파일을 빈 파일로 만든다.

등록부

Ext2의 등록부는 파일명과 이에 대응하는 i마디번호를 저장하는 자료블로크로 구성된 특별한 파일로 실현한다. 자료블로크는 ext2_dir_entry_2형태구조체로 이루어져있다. 이 구조체의 마당은 표 2-26에 있다. 이 구조체의 길이는 가변적이다. 마지막 name마당이 최대 EXT2_NAME_LEN개 문자(보통 255)로 이루어진 길이가 가변적인 배렬이기때문이다. 효률성을 위해 등록부입구점의 길이는 언제나 4의 배수며 필요하다면 파일명끝에 빈문자(《\0》)가 들어간다. name_len마당에 실제 파일명의 길이를 저장한다.

표 2-26. Ext2등록부항목의 마당

형	마 당	설 명
__u32	inode	i마디번호
__ u16	rec_len	등록부입구점의 길이
__u8	name_len	파일이름길이
__u8	file_type	파일류형
char [EXT2_NAME_LEN]	name	파일이름

file_type마당에는 파일류형을 나타내는 값을 저장한다. rec_len마당은 등록부의 유효한 다음 항목을 가리키는 지적자로 해석할수 있다. 유효한 다음 항목의 시작주소를 얻으려면 등록부항목에 이 편위값을 더하면 된다. 등록부항목을 삭제하려면 등록부항목의 i마디마당을 0으로 설정하고 유효한 이전 항목의 rec_len마당을 적절히 증가시키면 된다. rec_len마당을 자세히 보면 usr의 rec_len마당이 12+16(usr와 oldfile 입구점의 길이)으로 설정되여있으므로 oldfile항목이 삭제되였다는 사실을 알수 있다.

기호련결

앞에서 설명한것처럼 기호련결의 경로명이 60자이하면 4B정수 15개로 구성된 배렬인 i마디의 i_block마당에 기호련결을 저장하며 자료블로크를 사용하지 않는다.

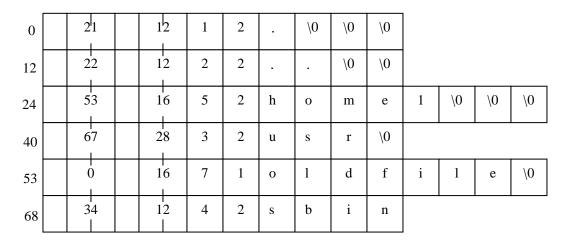
경로명이 60자이상이면 자료블로크하나를 사용한다.

장치파일, 관, 소케트

이런 파일에는 자료블로크가 필요없다. 모든 정보는 i마디에 저장된다.

0	21	12	1	2	.	\0	\0	\0				
12	22	12	2	2	.	.	\0	\0				
24	53	16	5	2	h	o	m	e	1	\0	\0	\0
40	67	28	3	2	u	s	r	\0				
53	0	16	7	1	o	l	d	f	i	l	e	\0
68	34	12	4	2	s	b	i	n				

그림 2-6. Ext2 등록부의 례

교육성 교육정보쎈터

3. Ext2기억기자료구조

효률성을 높이기 위해 파일체계를 탑재하면 Ext2구획의 디스크자료구조에 저장된 정보중 대부분을 RAM에 복사하여 핵심부가 디스크읽기연산을 여러번 수행하지 않도록 한다. 자료구조가 얼마나 자주 바뀌는지 대략적인 개념을 잡기 위해 몇가지 기본적인 연산을 살펴본다.

- 새로운 파일을 생성하면 Ext2초블로크의 s_free_inodes_count마당, 그리고 해당한 그룹서술자의 bg_free_inodes_count마당값을 감소시켜야 한다.
- 핵심부가 기존의 파일에 자료를 추가하여 이 파일에 할당된 자료블로크수가 증가하면 Ext2초블로크의 s_free_blocks_count마당과 그룹서술자의 bg_free_blocks_count마당값이 바뀌여야 한다.
- 기존의 파일의 일부를 다시 기록하는 경우에도 Ext2초블로크의 s_write마당을 갱신해야 한다.

모든 Ext2디스크자료구조를 Ext2구획의 블로크에 저장하기때문에 핵심부는 이런 자료구조를 최신상태로 유지하려고 완충기캐쉬와 페지캐쉬를 사용한다.

표 2-27은 Ext2파일체계와 파일에 관련한 각 자료류형에 대해 디스크에서 자료를 표현하기 위해 사용하는 자료구조, 기억기에서 핵심부가 사용하는 자료구조 그리고 얼마나 많은 캐쉬억을 사용하는지 결정하기 위한 간단한 규칙을 나타낸다. 자주 바뀌는 자료는 언제나 고속기억된다. 즉 자료는 항상 완충기캐쉬나 페지캐쉬에 저장되고 대응하는 Ext2구획이 탑재해제될 때까지 완충기캐쉬에 저장되여있다. 이를 위해 핵심부는 완충기의 사용계수기를 언제나 0보다 크게 유지한다.

표 2-27 Ext2자료구조와 VFS자료구조사이비교

류형	디스크자료구조	기억기자료구조	캐쉬방식
초블로크	ext2_super_block	ext2_sb_info	항상 캐쉬됨
그룹 서술자	ext2_group_desc	ext2_ group_desc	항상 캐쉬됨
블로크비트배치표	블로크에 저장된 비트배렬	완충기에 저장된 비트배렬	고정된 크기
i마디비트배치표	블로크에 저장된 비트배렬	완충기에 저장된비트배렬	고정된 크기
i마디	ext2_inode	ext2_inode_info	동적
자료 블로크	지정되지 않음	완충기 페지	동적
여유 i마디	ext2_inode	없음	캐쉬되지 않음
여유 블로크	지정되지 않음	없음	캐쉬되지 않음

캐쉬된적이 없는 자료는 의미있는 정보를 담고있지 않으므로 어떤 캐쉬에도 저장되지 않는다. 이 두 극단사이에 두가지 다른 방식 즉 고정된 제한방식과 동적방식이 있다. 고정된 제한(fixed limit)방식에서는 지정된 수의 자료구조만 완충기캐쉬에 저장할수 있다. 한계를 넘으면 오래된 자료구조는 디스크로 넘어간다. 동적(dynamic)방식에서 자료는 대응하는 객체(I마디 또는 블로크)를 사용하는 동안에만 완충기캐쉬에 있게 된다. 파일을 닫거나 자료블로크를 삭제하면 shrink_caches()함수가 대응하는 자료를 캐쉬에서 제거한다.

1) Ext2_sb_info와 ext2_inode_info구조체

Ext2파일체계를 탑재하면 파일체계고유의 표를 포함하고있는 VFS초블로크의 u마당을 ext2_sb_info구조체형태로 적재하여 핵심부가 전체적인 파일체계와 관련한 정보를 찾을수 있게 한다. 이 구조체는 다음과 같은 정보를 포함한다.

- 대부분의 디스크초블로크마당
- 디스크초블로크를 포함하는 완충기의 완충기머리부를 가리키는 지적자 s_sbh
- 디스크초블로크를 포함하는 완충기를 가리키는 지적자 s_es
- 한 블로크안에 들어갈수 있는 그룹서술자수 s_desc_per_block
- 그룹서술자를 포함하는 완충기의 완충기머리부배렬을 가리키는 지적자 s_group_desc(보통 항목 하나로 충분하다)
- 탑재상태, 탑재항목 등과 관련한 기타 자료
- 이와 류사하게 Ext2파일과 관련한 i마디객체를 초기화하면 ext2_inode_info 구조체형태로 u마당을 적재한다. 이 구조체는 다음과 같은 정보를 포함한다.
- 디스크의 i마디구조체에 있는 대부분의 마당중에서 일반 VFS i마디객체에 저장되지 않는 마당
- 단편크기와 단편번호(아직 사용되지 않음)
- i마디가 속한 블로크그룹의 색인 i_block_group
- 자료블로크를 미리 할당하기 위해 사용하는 i_allock_block 와 i_allock_count마당
- 디스크i마디를 동시에 갱신해야 하겠는가를 나타내는 기발인 i_osync마당

2) 비트배치표캐쉬

핵심부는 Ext2파일체계를 탑재하면 Ext2디스크초블로크를 위한 완충기를 할당하고 초블로크의 내용을 디스크에서 읽는다. 이 완충기는 Ext2파일체계를 탑재해제한 경우에만 해제된다. 핵심부가 Ext2초블로크의 마당내용을 변경해야 하는 경우 완충기의 적절한 위치에 새로운 값을 기록하고 완충기를 불결한것으로 표시한다.

그러나 모든 Ext2디스크자료구조에 대해서 이 접근방법을 사용할수는 없다. 최근 몇년동안 디스크용량이 10배이상 증가함에 따라 i마디와 자료블로크비트배치표의 크기도 10배이상 증가하였다. 따라서 동시에 모든 비트배치표를 RAM에 유지하는 일이 힘들어졌다.

례를 들어 크기가 1kB인 블로크를 사용하는 4GB디스크를 생각해보자. 각 비트배치표는 한 블로크의 모든 비트를 채우므로 각 비트배치표는 블로크 8192개의 상태 즉 8MB디스크공간을 나타낼수 있다. 따라서 필요한 블로크그룹의 수는 4096MB/8MB=512이다. 각 블로크그룹은 i마디비트배치표와 자료블로크비트배치표를 요구하므로 비트배치표 1024개를 모두 기억기에 저장하려면 1MB RAM이 필요하다.

Ext2서술자의 기억기요구를 제한하기 위해 모든 탑재된 Ext2파일체계에 대해 EXT2_MAX_GROUP_LOADED(보통 8)크기인 캐쉬 두개를 사용하는 방법을 채택하였다. 한 캐쉬에는 가장 최근에 접근한 i마디비트배치표를 저장하고 다른 캐쉬에는 가장 최근에 접근한 블로크비트배치표를 저장한다. 캐쉬에 포함된 비트배치표를 포함하는 완충기는 사용계수기값이 0보다 크기때문에 shrink_mmap()가 해제하지 않는다. 반대로 비트배치표캐쉬에 없는 비트배치표를 포함하는 완충기는 사용계수기값이 0이기때문에 여유기억기가 모자라면 해제된다. 매 캐쉬는 요소를 EXT2_MAX_GROUP_LOADED개 포함하는 배렬 두개를 사용하여 실현한다. 한 배렬에는 현재 캐쉬에 비트배치표가 있는 블로크그룹의 색인값을 저장하고 다른 배렬에는 비트배치표를 참조하는 완충기머리부를 가리키는 지적자를 저장한다.

ext2_sb_info구조체는 i마디비트배치표캐쉬와 관련한 배렬을 저장한다.

블로크그룹의 색인값은 s_inode_bitmap마당에, 완충기머리부에 대한 지적자는 s_inode_bitmap_number마당에 있다. s_inode_bitmap와s_block_number마당에는 블로크비트배치표캐쉬에 대응하는 배렬을 저장한다.

load_inode_bitmap()함수는 지정된 블로크그룹의 i마디비트배치표를 적재하고 비트배치표가 들어있는 캐쉬위치를 반환한다.

비트배치표가 비트배치표캐쉬에 없으면 load_inode_bitmap()는 read_inode_bitmap()를 호출한다. 이 함수는 그룹서술자의 bg_indoe_bitmap마당에서 비트배치표를 포함하는 블로크수를 얻은 다음 bread()를 호출하여 새로운 완충기를 할당한다. 완충기캐쉬에 아직 없다면 디스크에서 블로크를 읽는다.

Ext2구획의 블로크그룹수가 EXT2_MAX_GROUP_LOADED보다 작거나 같으면 비트배치표를 삽입할 캐쉬배렬위치색인값은 항상 load_inode_bitmap()함수에 변수로 넘긴 블로크그룹색인값과 일치한다. 반면에 캐쉬공간보다 더 많은 블로크그룹이 있다면 LRU(Least Recently Used)기법을 사용해서 캐쉬에서 비트배치표를 제거하고 요청한 비트배치표를 첫번째 캐쉬위치에 넣는다. 그림 2-7은 블로크그룹 5에 있는 비트배치표를 참조하는 3가지 경우를 보여준다. 요청한 비트배치표가 이미 캐쉬에 있는 경우 캐쉬에 없지만 여유공간이 있는 경우 그리고 캐쉬에도 없고 여유공간도 없는 경우이다.

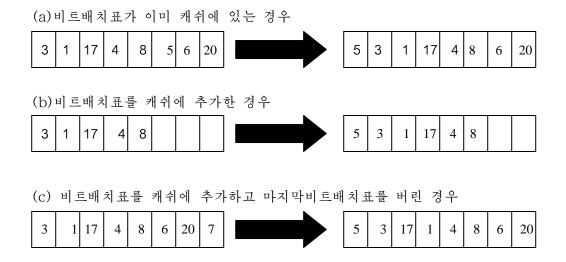



그림 2-7. 캐쉬에 비트배치표 추가하기

load_block_bitmap()와 read_block_bitmap()함수는 load_inode_bitmap()와 read_inode_bitmap()함수와 아주 류사하지만 Ext2구획의 블로크비트배치표캐쉬를 참조한다.

그림 2-8은 탑재된 Ext2파일체계의 기억기자료구조를 나타낸다. 실례에는 디스크의 세 블로크안에 서술자를 저장한 세 블로크그룹이 있다. 따라서 ext2_sb_info의 s_group_desc마당은 완충기머리부의 세개의 배렬을 가리킨다. 여기서는 색인값이 2인 i마디비트배치표하나와 색인값이 4인 블로크비트배치표 하나만 볼수 있다. 핵심부는 비트배치표캐쉬에 비트배치표 2*EXT2_MAX_GROUP_LOADED개를 저아할수 있으며 더 많은 비트배치표를 완충기캐쉬에 저장할수 있다.

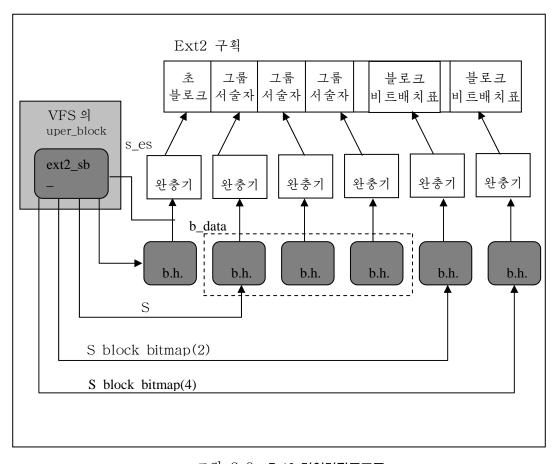



그림 2-8. **Ext2 기억기자료구조**

4. Ext2파일체계생성

디스크에 파일체계를 생성하는 작업은 크게 두 단계로 이루어진다. 첫번째 단계는 디스크를 형식화하는것으로서 디스크장치구동프로그람이 디스크에 블로크를 읽고 쓸수 있게 한다. 최신 하드디스크는 공장에서 형식화해서 나오므로 다시 형식화할 필요가 없다. 유연성자기원판은 Linux에서 /usr/bin/superformat 유틸리티프로그람을 사용하여 형식화할수 있다. 두번째 단계는 파일체계를 생성하는것으로서 이 절의 앞부분에서 설명한 구조체를 설정한다.

/sbin/mke2fs유틸리티프로그람을 사용하여 Ext2파일체계를 생성할수 있다. 이 프로그람은 다음과 같은 기본선택항목들을 가정한다. 사용자는 선택항목을 변경하기 위해 명령에 기발을 사용할수 있다.

- 블로크크기 : 1024B
- 단편크기 : 블로크크기(블로크단편화는 실현되지 않았다.)

- 할당된 i마디개수 : 4096B당 i마디 한개
- 예약된 블로크의 비률 : 5%

이 프로그람은 다음과 같은 작업을 수행한다.

① 초블로크와 그룹서술자를 초기화한다.

② 구획에 손상된 블로크가 있는지 검사한다.(선택항목) 손상된 블로크가 있다면 손상된 블로크목록을 만든다.

③ 매 블로크그룹에 대해 초블로크, 그룹서술자, i마디표, 비트배치표 두개를 저장하는데 필요한 디스크블로크를 예약한다.

④ 매 블로크그룹의 i마디비트배치표와 자료블로크비트배치표를 0으로 초기화한다.

⑤ 매 블로크그룹의 i마디표를 초기화한다.

⑥ 뿌리등록부 /를 생성한다.

⑦ Lost+found등록부를 생성한다. e2fsck는 없어지거나 손상된 블로크를 련결하기 위해 이 등록부를 사용한다.

⑧ 앞에서 만든 두 등록부가 속한 블로크그룹의 i마디비트배치표와 자료블로크비트배치표를 갱신한다.

⑨ 손상된 블로크가 없으면 lost+fount등록부로 옮긴다.

mke2fs가 기본선택항목으로 1.4MB유연성디스크를 Ext2파일체계로 어떻게 초기화하는지 살펴보자.

탑재되면 이 유연성자기원판은 VFS에서 볼 때 길이가 1024B인 블로크 1390개로 구성된 볼륨 하나로 보인다. 디스크내용을 검사하기 위해 다음 Unix명령을 사용하여 유연성자기원판의 내용의 16진수쏟기(dump)가 들어있는 파일 dump_hex를 /tmp등록부에 생성할수 있다.

```
$ dd if=/dev/fd0 bs=1k count=1440 | od   tx1   Ax > /tmp/dump_hex
```

이 파일을 살펴보면 디스크용량의 제한때문에 그룹서술자 하나로 충분하다는 사실을 알수 있다. 또한 예약된 블로크개수가 72(1440의 5%0)이라는 사실과 기본선택항목에 따라 i마디표에는 4096B마다 i마디하나씩을 포함해야 하므로 블로크45개에 i마디 360개가 저장되여있다는 사실을 알수 있다.

표 2-28에 기본선택항목을 사용했을 때 유연성자기원판에 Ext2 파일체계를 어떻게 만드는지 요약하였다.

표 2-28 유연성자기원판에 대한 Ext2블로크할당

블로크	내 용
0	기동블로크
1	초블로크
2	블로크그룹서술자 하나를 포함한 블로크

교육성 교육정보쎈터

3	자료블로크 비트배치표
4	i마디비트배치표
5-49	i마디표 : 10번까지 예약됨, 11번은 lost+found, 12-360은 여유블로크
50	뿌리등록부(., .., lost +found 포함)
51	lost+found 등록부 (.과 ..을 포함)
52-62	lost+found 등록부에 미리할당된 예약블로크 여유블로크
63-1439	여유블로크

5. Ext2메쏘드

대부분의 VFS메쏘드에는 대응하는 Ext2실현이 있다. 디스크와 기억기자료구조를 완전히 리해하면 이것을 실현하고있는 Ext2함수코드로 따라 들어갈수 있다.

1) Ext2초블로크연산

대부분의 VFS초블로크연산 즉 read_inode, write_inode, put_inode, delete_inode, put_super, write_super, statfs, remount_fs 등은 Ext2에서 독자적으로 실현된다. 초블로크메쏘드의 주소는 지적자의 배렬인 ext2_sops에 저장된다.

2) Ext2 i마디연산

어떤 VFS색인마디연산은 Ext2에서 독자적으로 실현되는데 그 실현은 i마디가 참조하는 파일류형에 따라 다르다.

i마디가 정규파일을 나타내는 경우 truncate연산만 ext2_truncate() 함수로 실현하고 ext2_file_inode_operations표에 있는 다른 모든 i마디연산은 NULL지적자를 가진다. 대응하는 Ext2메쏘드가 정의되여있지 않은 경우(NULL지적자) VFS는 자신의 범용함수를 사용한다.

i마디가 등록부를 나타내는 경우 ext2_dir_inode_operations표에 있는 대부분의 i마디연산은 독자적인 Ext2함수로 실현한다.(표 2-29 참고)

표 2-29 . 등록부파일에 대한 Ext2색인마디연산

VFS i마디연산	Ext2 등록부 i마디 메쏘드
create	ext2_ create()
lookup	ext2_lookup()
link	ext2_link()
unlink	ext2_ symlink()
mkdir	ext2_mkdir()
rmdir	ext2_rmdir()
mknod	ext2_mknod()
rename	ext2_rename()

i마디가 자체에 저장할수 있는 기호련결을 나타내는 경우(짧은 기호련결),

readlink와 follow_link메쏘드를 제외한 모든 i마디메쏘드는 NULL이고 두 메쏘드는 각각 ext2_readlink()와 ext2_follow_link()함수가 실현한다. 이 메쏘드들의 주소는 ext2_fast_symlink_inode_operations표에 저장된다. i마디가 자료블로크에 저장되는 긴 기호련결을 나타내는 경우 readlink와 follow_link메쏘드는 범용 page_readlink()와 page_follow_link()함수로 실현되며 이것들의 주소는 page_symlink_inode_operations표에 저장된다.

i마디가 문자장치파일, 블로크장치파일, 이름있는 관을 나타내는 경우 i마디연산은 파일체계에 의존하지 않는다. 이것들은 각각 chrdev_inode_operations, blkdev_inode_operations, fifo_inode_operations표에 저장된다.

3) Ext2파일연산

Ext2파일체계의 독자적인 파일연산은 표 9에 있다. 표에서 볼수 있는것처럼 VFS 메쏘드중에는 여러 파일체계에 공통적인 범용함수로 실현하는것도 있다. 이 메쏘드들의 주소는 ext2_file_operations표에 저장된다.

표 2-30 Ext2파일연산

VFS 파일 연산	Ext2메쏘드
llseek	generic_ file_llseek ()
read	generic_ file_ read()
write	generic_ file_write ()
ioctl	ext2_ioctl()
mmap	generic_ file_ mmap()
open	generic_ file_ open()
release	ext2_release_file()
fsync	ext2_sync_file()

Ext2의 read와 write메쏘드는 각각 generic_file_read() 와 generic_file_write()함수로 실현하는데 이 함수는 2절에 있는 《파일에서 읽기》와 《파일에 쓰기》에서 설명하였다.

6. Ext2디스크공간관리

파일을 디스크에 실제로 저장하는 방식은 프로그람작성자가 파일을 보는 관점과 다른 면이 있다. 블로크는 디스크에 흩어져있을수 있고(비록 파일체계는 접근시간을 향상하기 위해 블로크를 련속적인 상태로 유지하려고 하지만) 프로그람이 (lseek()체계호출을 사용하여) 파일에 구멍을 만들수 있으므로 파일이 프로그람작성자에게 실제보다 더

크게 보일수 있다.

여기서는 Ext2파일체계가 디스크공간을 관리하는 방법 즉 i마디와 자료블로크를 할당하고 해제하는 방법을 설명한다. 다음 두가지 문제가 특히 중요하다.

・공간관리에서 가능하면 반드시 파일단편화(file fragmentation)를 피해야 한다. 파일단편화는 련속하지 않는 디스크블로크에 있는 여러 작은 스랩에 파일의 물리적저장공간이 흩어져 저장되는것을 의미한다. 파일단편화는 읽기연산중에 디스크머리부위치를 자주 바꿔야 하므로 파일에 대한 순차적읽기연산의 평균시간을 증가시킨다.

・공간관리는 반드시 빨리 동작해야 한다. 핵심부는 파일편위에서 여기에 대응하는 Ext2구획의 론리블로크번호를 아주 빨리 얻을수 있어야 한다. 이렇게 하려면 핵심부는 디스크에 저장된 주소표에 접근하는 회수를 최대한 제한해야 한다. 이런 접근은 평균파일접근시간을 상당히 늘이기때문이다.

1) i마디생성

Ext2_new_inode()함수는 Ext2디스크i마디를 생성하고 대응하는 i마디객체의 주소(실패한 경우 NULL)를 반환한다. 이 함수는 새로운 i마디가 삽입될 등록부를 가리키는 i마디객체의 주소인 dir와 생성할 i마디류형을 나타내는 mode를 변수로 사용한다. mode는 i마디가 할당될 때까지 현재프로세스를 연기하도록 요청하는 MS_SYNCHRONOUS기발을 포함한다. 이 함수는 다음과 같은 작업을 실행한다.

① new_inode()를 호출하여 새로운 i마디객체를 할당하고 이 객체의 i_sb마당을 dir->i_sb에 저장된 초블로크주소로 설정한다.

② 부모초블로크에 있는 s_lock신호기에 대해 down()을 호출한다. 신호기가 이미 사용중이면 핵심부는 현재프로세스를 연기한다.

③ 새로운 i마디가 등록부이면 다 채워지지 않은 블로크그룹에 등록부가 골고루 흩어지도록 저장하려고 시도한다. 특히 평균보다 많은 여유 i마디가 있는 모든 블로크그룹중에서 가장 많은 여유블로크가 있는 블로크그룹에 새로운 등록부를 할당한다.(평균은 전체 여유i마디수를 블로크그룹수로 나눈값이다.)

④ 새로운 i마디가 등록부가 아니면 여유i마디가 있는 블로크그룹에 이 i마디를 할당한다. 이 함수는 부모등록부가 들어있는 그룹에서 시작하여 점점 더 멀이지는 방향으로 이동한다. 자세히 설명하면 다음과 같다.

ㄱ. 부모등록부 dir가 들어있는 블로크그룹에서 시작하여 간단한 로그탐색을 실행한다. 알고리듬은 블로크그룹 log(n)개를 탐색한다. 여기서 n은 전체블로크그룹의 개수다. 알고리듬은 활용할수 있는 블로크그룹을 발견할 때까지 다음과 같이 건너뛴다. 시작하는 블로크그룹번호를 i라 하면 알고리듬은 블로크그룹 i mod (n), i+1 mod (n), i+1+2 mod (n), i+1+2+4 mod (n),…을 탐색한다.

ㄴ. 로그탐색에서 여유i마디가 있는 블로크그룹을 찾지 못하면 함수는 부모등록부

가 들어있는 블로크그룹부터 시작해서 모든 블로크그룹을 차례로 탐색한다.

⑤ load_inode_bitmap()를 호출하여 선택한 블로크그룹의 i마디비트배치표를 얻고 그 안에서 맨 처음 NULL비트를 찾아서 첫번째 여유디스크 i마디의 번호를 얻는다.

⑥ 디스크i마디를 할당한다. i마디비트배치표에서 i마디에 대응하는 비트를 1로 설정하고 비트배치표를 포함하는 완충기를 불결한것으로 표시한다. 그리고 파일체계를 탑재할 때 MS_SYNCHRONOUS기발을 지정하였다면 ll_rw_block()를 호출하고 쓰기연산이 완료될 때까지 기다린다.

⑦ 그룹서술자의 bg_free_inodes_count마당을 감소시킨다. 새로운 i마디가 등록부이면 bg_used_dirs_count를 증가시킨다. 그룹서술자를 포함하는 완충기를 불결한것으로 표시한다.

⑧ 디스크초블로크의 s_free_inodes_count마당을 감소시키고 이 초블로크를 포함하는 완충기를 불결한것으로 표시한다. VFS의 초블로크객체의 s_dirs마당을 1로 설정한다.

⑨ i마디객체의 마당을 초기화한다. 특히 i마디번호 i_ino를 설정하고 xtime.rv_sec값을 i_atime, i_mtime, i_ctime에 복사한다. 그리고 ext2_inode_info구조체의 i_block_group마당을 블로크그룹 색인값으로 채운다. 이 마당의 의미는 표 3를 참고하면 된다.

⑩ 새로운 i마디객체를 하쉬표 inode_hashtable에 삽입한다. 그리고 mark_inode_dirty()를 호출하여 i마디객체를 초블로크의 불결한 i마디목록으로 옮긴다.

⑪ 부모초블로크에 있는 s_lock신호기에 대해 up()을 호출한다.

⑫ 새로운 i마디객체의 주소를 반환한다.

2) i마디제거

ext2_free_inode()함수는 변수로 주소를 넘긴 i마디객체가 가리키는 디스크i마디를 지운다. 핵심부는 이 함수를 호출하기 전에 내부자료구조와 파일자체의 자료를 정리하는 연산을 수행해야 한다. 또한 i마디하쉬표에서 i마디객체를 지운 후 혹은 자신의 등록부에서 i마디를 가리키던 마지막 하드련결을 삭제한 다음 그리고 파일의 모든 자료블로크를 제거하기 위해 파일의 길이를 0으로 만든 다음에는 반드시 이 함수를 호출해야 한다.

이 함수는 다음 작업을 실행한다.

① 부모초블로크에 있는 s_lock신호기에 대해 down()을 호출하여 초블로크객체에 대한 배타적인 접근권한을 얻는다.

② clear_inode()를 호출하여 다음 연산을 수행한다.

ㄱ. invalidate_inode_buffers()를 호출하여 i마디에 속한 불결한 완충기들을 i마디의 i_dirty_buffers와 i_dirty_data_buffers 목록에서 제거한다.

ㄴ. i마디의 I_LOCK기발이 설정되여있으면 i마디의 완충기중에 입출력자료전송에 관

련한것이 있는것이다. 함수는 입출력자료전송이 완료될 때까지 현재프로세스를 연기한다.

ㄷ. 초블로크객체의 clear_inode메쏘드가 정의되여있으면 이 메쏘드를 호출한다. Ext2파일체계는 이 메쏘드를 정의하고있지 않다.

ㄹ. i마디의 상태를 I_CLEAR로 설정한다.(i마디객체의 내용은 이제 의미가 없다.)

③ i마디번호와 각 블로크그룹의 i마디수로부터 디스크i마디가 있는 블로크그룹의 색인값을 계산한다.

④ i마디비트배치표를 얻기 위해 load_inode_bitmap()를 호출한다.

⑤ 그룹서술자의 bg_free_inodes_count마당을 증가시킨다. 삭제된 i마디가 등록부이면 bg_used_dirs_count마당을 감소시킨다. 그룹서술자가 들어있는 완충기를 불결한것으로 표시한다.

⑥ 디스크초블로크의 s_free_inodes_count마당을 증가시키고 초블로크가 들어있는 완충기를 불결한것으로 표시한다. 초블로크객체의 s_dirs마당을 1로 설정한다.

⑦ i마디비트배치표에서 디스크i마디에 대응하는 비트를 지우고 비트배치표가 들어있는 완충기를 불결한것으로 표시한다. 그리고 파일체계를 탑재할 때 MS_SYNCHRONOUS기발을 지정하였다면 ll_rw_block()를 호출하고 비트배치표완충기에 대한 쓰기연산이 끝날 때까지 기다린다.

⑧ 부모초블로크객체에 들어있는 s_lock신호기에 대해 up()을 호출한다.

3) 자료블로크의 주소지정

비여있지 않은 정규파일은 여러 자료블로크그룹으로 이루어져있다. 이 블로크는 파일내의 상대적인 위치나(파일블로크번호) 디스크구획내의 위치(론리블로크번호)를 사용하여 참조할수 있다. 파일내의 상대적위치인 편위 f로부터 여기에 대응하는 자료블로크의 론리블로크번호를 얻는 과정은 두 단계로 이루어진다.

- 편위f로부터 파일블로크번호 즉 편위f에 있는 문자를 포함하는 블로크의 색인값을 얻는다.
- 파일블로크번호를 여기에 대응하는 론리블로크번호로 변환한다.

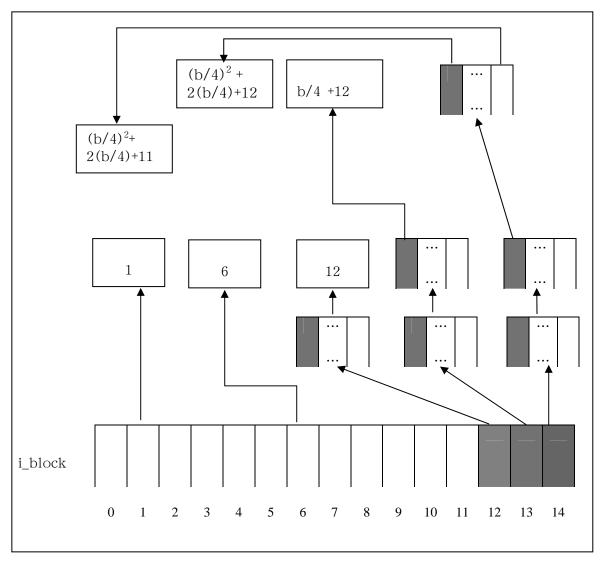
Unix파일은 조종문자를 포함하지 않으므로 파일의 f번째 문자를 포함하는 파일블로크번호를 얻는 일은 아주 쉽다. f를 파일체계의 블로크크기로 나눈 결과로부터 이보다 작은 가장 가까운 정수를 취한다.

례를 들어 블로크크기가 4kB라고 가정하자. f가 4096보다 작으면 문자는 파일의 첫번째 자료블로크에 있을것이고 이 블로크의 파일블로크 번호는 0이다. f가 4096보다 크거나 같고 8192보다 작으면 문자는 파일블로크번호 1인 자료블로크에 포함되여있으며 이런 식으로 계속된다.

파일블로크번호를 구하는것은 아주 쉽다. 그렇지만 파일블로크번호를 이에 대응하는 론리블로크번호로 변환하는 일은 말처럼 그리 쉽지 않다. 디스크에서 Ext2파일의 자료

블로크가 반드시 련속되지는 않기때문이다.

따라서 Ext2파일체계는 각 파일블로크번호와 이에 대응하는 론리블로크번호사이의 관계를 디스크에 저장하는 방법을 제공해야 한다. 이 배치는 AT&T의 초기Unix판본까지 거슬러 올라가는데 i마디내부에서 부분적으로 실현한다. 또한 큰 파일을 처리하는데 사용하는 i마디확장인 여분의 지적자를 포함하는 특수자료블로크와도 관련있다.

디스크i마디의 i_block마당은 론리블로크번호를 담고있는 구성요소 EXT2_N_BLOCKS개의 배렬이다. 여기서는 EXT2_N_BLOCKS의 기본값이 15라고 가정한다. 배렬은 그림 2-9에서 설명한 큰 자료구조의 처음 부분을 나타낸다. 그림에서 보는것처럼 배렬의 요소 15개는 4가지 류형으로 되여있다.

2-9. 파일의 자료 블로크 주소를 얻는데 사용하는 자료구조

- 처음 12개 요소는 파일의 처음 12개 블로크, 즉 파일블로크번호 0에서 11까지의 블로크에 대응하는 론리블로크번호를 나타낸다.
- 12번요소는 론리블로크번호의 2차배렬을 나타내는 블로크의 론리블로크번호를 포함한다. 즉 12에서 b/4+11까지의 파일블로크번호에 대응한다. 여기서 b는 파일체계의 블로크크기이다.(매 론리블로크번호는 4B를 차지하므로 식에서 4로 나눈다.) 따라서 핵심부는 이 요소에서 블로크를 가리키는 지적자를 살펴보고 해당 블로크에서 다시 파일내용을 저장하고있는 최종블로크에 대한 지적자를 얻을수 있다.
- 13번요소는 론리블로크번호의 2차배렬을 포함하는 블로크의 론리블로크번호를 포함한다. 다음으로 이 2차배렬의 입구점은 3차배렬을 가리키고 3차배렬은 b/4+12에서 $(b/4)^2 + (b/4)+11$ 범위의 파일블로크번호에 대응하는 론리블로크번호를 포함한다.
- 마지막으로 14번요소는 중간단계를 세번 거친다. 4차배렬은 $(b/4)^2 + (b/4)+12$에서 $(b/4)^3 + (b/4)^2 + (b/4)+11$범위의 파일블로크번호에 대응하는 론리블로크번호를 저장한다.

이 기구는 작은 파일에 더 유리하다. 파일이 자료블로크를 12개이상 요구하지 않으면 디스크에 두번만 접근하면 모든 자료를 얻을수 있다. 한번은 디스크i마디의 i_block 배렬의 요소를 읽기 위해 다른 한번은 요청한 자료블로크를 읽기 위해서이다. 더 큰 파일일 경우 요청한 블로크를 읽으려면 련속적인 디스크접근 세번 또는 네번이 필요할수도 있다. 이것은 사실 최악의 경우에 대한 예측인데 실제로는 i마디, 완충기, 페지캐쉬 등이 디스크접근회수를 상당히 감소시키기때문이다.

파일체계의 블로크크기도 이 기구에 영향을 준다. 블로크크기가 크면 한 블로크안에 론리블로크번호를 더 많이 저장할수 있다. 표 2-31에서는 각 블로크크기와 주소지정방식에서 가능한 파일크기의 최대값을 보여준다. 례를 들어 블로크크기가 1024B이고 파일에 자료가 268kB있으면 직접배치를 통해 파일의 처음 12kB에 접근하고 나머지 13-268kB는 간접배치를 통하면 된다. 2GB보다 큰 파일을 32bit방식에서 사용하려면 파일을 열 때 O_LARGEFILE항목을 지정해야 한다. Ext2파일체계의 파일크기한계는 2TB-4096B이다.

표 2-31. 1자료블로크주소지정에 대한 파일크기의 최대값

블로크 크기	직접	1- 간접	2-간접
1024	12kB	268kB	64.26MB
2048	24kB	1.02MB	513.02MB
4096	48kB	4.04MB	4GB

3) 파일구멍

파일구멍(file hole)은 정규파일의 일부분으로 null문자만 포함하며 디스크에는 여기에 대응하여 저장된 자료블로크가 없는 부분이다. 파일구멍은 오래된 Unix파일의 특징중 하나이다. 례를 들어 다음과 같은 Unix명령은 앞부분에 구멍이 몇바이트 있는 파일을 생성한다.

$ echo n "X" | dd of=/tmp/hole bs=1024 seek=6

이제 /tmp/hole에는 문자가 6145개 있지만(null문자 6144개와 X문자 하나), 파일은 디스크에서 자료블로크 하나만 차지한다.

파일구멍은 디스크공간의 랑비를 막으려고 도입한것이다. 주로 자료기지응용프로그람에서 사용하며 파일에 대해 하쉬처리를 하는 모든 응용프로그람에서 사용한다.

Ext2의 파일구멍실현은 동적자료블로크할당을 기반으로 한다. 프로쎄스가 자료를 기록해야 할 때에만 실제로 파일에 블로크를 할당한다. 매 i마디의 i_size마당은 구멍을 포함하여 프로그람에 보여주는 전체 파일크기를 정의하지만 i_blocks마당은 실제로 파일에 할당된 자료블로크수를 저장한다.(단위는 512B이다.)

앞의 dd명령실례에서 블로크크기가 4096인 Ext2구획에 /tmp/hole 파일을 생성하였다고 하자. 대응하는 디스크i마디의 i_size마당에는 6145를 저장하고 i_blocks마당에는 8을 저장한다.(각 4096B블로크는 512B블로크 8개를 저장할수 있다.) i_block배렬의 두번째 요소(파일블로크번호 1인 블로크에 대응하는)는 할당된 블로크의 론리블로크번호를 저장하고 배렬내에 있는 나머지 요소는 모두 null이다.(그림 2-10참고)

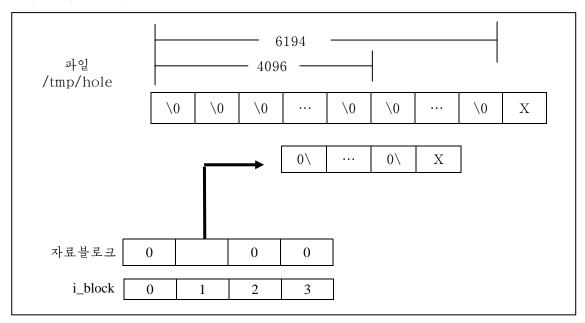



그림 2-10. 앞부분에 구멍이 있는 파일

4) 자료블로크할당

핵심부는 Ext2정규파일 자료를 저장하고있는 블로크를 찾아야 할 때 ext2_get_blk()함수를 호출한다. 이 함수는 해당 블로크가 없으면 자동으로 블로크를 할당한다. 이 함수는 핵심부가 Ext2정규파일에 대한 읽기나 쓰기연산을 요청할 때마다 호출된다.

ext2_get_block()함수는 《자료블로크의 주소지정》에서 설명한 자료구조를 처리하며 필요하면 ext2_alloc_block()함수를 호출하여 Ext2 구획에서 여유블로크를 찾는다.

Ext2파일체계는 파일단편화를 줄이기 위해 파일에서 마지막으로 할당된 블로크근처에 새로운 블로크를 할당하려고 한다. 실패하면 파일체계는 파일의 i마디가 포함된 블로크그룹안에서 새로운 블로크를 탐색한다. 마지막으로 다른 블로크그룹에서 여유블로크를 구한다.

Ext2파일체계는 자료블로크를 미리 할당한다. 파일은 요청한 블로크뿐만 아니라 최대 8개까지 린접한 블로크그룹을 얻는다. ext2_inode_info구조체의 i_prealloc_count마당은 미리 할당된 자료블로크중에서 아직 사용하지 않은 블로크수를 나타내며 i_prealloc_block마당은 다음에 사용할 블로크의 론리블로크번호를 나타낸다. 미리 할당된 블로크중에 사용하지 않은 블로크는 파일을 닫을 때 파일크기를 0으로 할 때 또는 쓰기연산이 블로크를 미리 할당하도록 한 쓰기연산에 대해 련속하지 않으면 해제된다.

ext2_allock_block()함수는 i마디객체에 대한 지적자와 목적지(goal)를 변수로 받는다. 《목적지》는 새로운 블로크가 선택하는 위치를 나타내는 론리블로크번호이다. ext2_getblk()함수는 다음과 같은 과정을 거쳐 goal변수를 설정한다.

① 할당하려는 블로크와이전에 할당된 블로크의 파일블로크번호가 련속하면 goal은 이전 블로크의 론리블로크번호+1이다. 프로그람에서 보이는 련속된 블로크를 디스크에서도 련속하게 한다.

② 첫번째규칙이 적합하지 않고 파일에 적어도 한 블로크를 할당했으면 목적지는 이것들 블로크의 론리블로크번호중 하나이다. 더 정확하게는 블로크를 할당할 파일에 이미 할당된 블로크의 론리블로크번호이다.

③ 앞의 규칙을 모두 적용할수 없으면 목적지는 파일의 i마디를 포함하는 블로크그룹내 첫번째 블로크(여유블로크일 필요는 없음)의 론리블로크번호이다.

ext2_alloc_block()함수는 목적지가 미리 할당된 블로크중 하나인지 검사한다. 그렇다면 대응하는 블로크와 블로크의 론리블로크번호를 반환한다. 그렇지 않으면 함수는 미리 할당된 상태로 남아있는 모든 블로크를 해제하고 ext2_new_block()를 호출한다.

이 함수는 다음과 같은 전략을 사용하여 Ext2구획내의 여유블로크를 탐색한다.

① ext2_alloc_block()에 전달하려고 선택한 블로크(goal)가 여유블로크이면 이것을 할당한다.

② 목적지가 사용중이면 해당 블로크다음의 블로크 64개중에서 여유블로크가 있는

지 검사한다.

③ 해당 블로크다음의 린접한 령역에서 여유블로크를 찾지 못했으면 goal을 포함하는 블로크그룹에서 시작하여 모든 블로크그룹을 찾는다. 각 블로크그룹에 대해 함수는 다음을 실행한다.

ㄱ. 최소한 여유블로크 8개가 린접하는 그룹을 찾는다.

ㄴ. 이와 같은 블로크가 없으면 여유블로크 하나를 찾는다.

여유블로크를 찾으면 탐색을 끝낸다. 완료하기 전에 ext2_new_block() 함수는 찾은 여유블로크와 린접한 여유블로크를 8개까지 미리 할당하려고 시도하고 디스크i마디의 i_prealloc_block와 i_prealloc_count마당을 적절한 블로크위치와 블로크수로 설정한다.

5) **자료블로크해제**

프로쎄스가 파일을 삭제하거나 파일길이를 0으로 만들면 이 파일의 모든 자료블로크를 반환해야 한다. 이 작업은 ext2_truncate()로 처리하는데 파일i마디객체의 주소를 변수로 받는다. 이 함수는 디스크i마디의 i_block배렬을 차례로 읽어서 모든 자료블로크를 찾고 간접참조에 사용된 모든 블로크를 찾는다. 그리고 매 그룹을 ext2_free_blocks()를 호출하여 해제한다.

ext2_free_blocks()함수는 린접한 자료블로크 하나이상의 그룹을 해제한다. ext2_truncate()에서 사용하는것외에 이 함수는 미리 할당된 파일블로크를 해제할 때 호출된다. 이 함수의 변수는 다음과 같다.

inode

파일을 나타내는 i마디객체주소

block

해제할 첫번째 블로크의 론리블로크번호

count

해제할 린접블로크수

함수는 초블로크의 신호기 s_lock에 대해 down()을 호출하여 파일체계의 초블로크에 대한 배타적인 접근 권한을 얻은 다음 해제할 각 블로크에 대해 다음을 수행한다.

① 해제할 블로크를 포함하는 블로크그룹의 블로크비트배치표를 얻는다.

② 해제할 블로크에 대응하는 블로크비티배치표의 비트를 지우고 비트배치표를 포함하는 완충기를 불결한것으로 표시한다.

③ 블로크그룹서술자의 bg_free_blocks_count마당을 증가시키고 대응하는 완충기를 불결한것으로 표시한다.

④ 디스크초블로크의 s_free_blocks_count마당을 증가시키고 대응하는 완충기를 불결한것으로 표시하며 초블로크객체의 s_dirty기발을 설정한다.

⑤ MS_SYNCHRONOUS기발을 설정하여 파일체계를 탑재했으면 ll_rw_block()를 호출하고 비트배치표의 완충기에 대한 쓰기연산을 끝낼 때까지 기다린다.

이 함수는 마지막으로 up()을 호출하여 초블로크의 s_lock신호기를 해제한다.

7. Ext3파일체계

여기서는 Ext2에서 발전한 파일체계인 Ext3를 설명한다. 새로운 파일체계는 다음 두가지 개념을 기반으로 설계되였다.

- 기록형파일체계를 제공한다.
- 가능하면 최대로 이전 Ext2파일체계와 호환성을 유지한다.

Ext3는 두 목표를 모두 잘 달성하고있다. 특히 Ext2에 기반을 두고있어서 디스크에서의 자료구조는 기본적으로 Ext2파일체계와 같다. 어떤 Ext3파일체계를 탑재해제하고 Ext2파일체계를 다시 탑재할수 있다. 또 Ext2파일체계에 기록(journal)을 생성하고 Ext3파일체계로 다시 탑재하는것도 간단하다.

Ext3와 Ext2사이의 호환성으로 하여 앞부분에서 설명한 대부분의 내용을 Ext3에 그대로 적용할수 있다. 따라서 여기에서는 Ext3가 제공하는 새로운 기능, 즉 기록형기능을 중심으로 설명한다.

1) 기록형파일체계

전통적인 Unix파일체계(Ext2를 포함)의 설계단계의 결정중에서 디스크용량이 증가함에 따라 바꾸어야 하는것들이 있게 되였다. 파일체계블로크에 대한 갱신이 디스크에 흘리기되기 전에 동적기억기에 오래동안 남아있을수 있다. 전원이 나가거나 체계에 장애가 발생한 경우 파일체계의 일관성이 파괴된 상태가 될수 있다. 이 문제를 해결하기 위해 전통적인 Unix 파일체계를 탑재하기 전에 파일체계검사를 수행한다. 제대로 탑재해제되지 않았으면 어떤 프로그람이 실행되여 디스크에 있는 전체 파일체계의 자료구조를 검사하고 수정하는 시간이 오래 걸리는 작업을 수행한다.

례를 들어 Ext2파일체계의 상태는 디스크초블로크의 s_mount_state 마당에 저장된다. 기동스크립트는 e2fsck유틸리티프로그람을 호출하여 이 마당에 저장된 값을 검사한다. 이 값이 EXT2_VALID_FS가 아니면 파일체계가 제대로 탑재해제되지 않은것이고 e2fsck는 파일체계의 전체 디스크자료구조에 대한 검사를 시작한다.

물론 파일체계의 일관성을 검사하기 위해 필요한 시간은 주로 검사할 파일과 등록부의 수에 따라 다시 말해서 디스크크기에 따라 결정된다. 수백GB에 이르는 파일체계인 경우에는 한번 검사하는데 몇시간이 걸리수 있다. 그리고 이런 몇시간이라는 정지된 시간(down time)은 상용환경이나 고가용성(high availability)봉사기에서 허용할수 없는 수준이다.

기록형파일체계의 목표는 최근의 디스크쓰기연산만을 가지고있는 특별한 디스크령역인 기록(journal)을 리용함으로써 오랜 시간이 필요한 전체 파일체계에 대한 일관성검사를 피해보자는것이다. 보통 체계에 장애가 발생했을 때 기록형파일체계를 다시 탑재하는데 걸리는 시간은 수s정도이다.

2) Ext3기록형파일체계

Ext3기록형의 사상은 파일체계에 대한 모든 고수준변경을 두 단계를 통해 실행하는 것이다. 첫번째 단계에서 디스크에 기록할 블로크의 복사본을 기록에 저장한다. 그리고 기록에 대한 입출력자료전송이 완료되면(자료가 기록에 위임되면) 블로크를 파일체계에 기록한다. 파일체계에 대한 입출력자료전송이 완료되면(자료가 파일체계에 위임되면) 기록에 있는 블로크복사본을 제거한다.

체계에 장애가 발생했을 때 복구하는 과정에서 e2fsck프로그람은 두가지 경우를 구분한다.

기록에 위임되기 전에 발생한 체계장애. 고수준변경을 나타내는 블로크의 복사본이 기록에 없거나 완료되지 않았다. 이런 경우 e2fsck는 이것들을 무시한다.

기록에 위임된 다음에 발생한 체계장애. 블로크의 복사본은 유효하며 e2fsck는 이것들을 파일체계에 기록한다.

첫번째 경우 파일체계에 대한 고수준변경은 사라지지만 파일체계의 상태의 일관성은 유지된다. 두번째 경우 e2fsck는 전체 고수준변경을 적용하므로 파일체계에 대한 입출력자료전송이 제대로 완료되지 않아서 발생하는 일관성이 파괴된 상태를 해결한다.

하지만 기록형파일체계에 너무 많은것을 기대하면 안된다. 기록형기능은 체계호출수준에서만 일관성을 보장한다. 례를 들어 write()체계호출을 여러번 사용하여 큰 파일을 복사하는 도중 체계장애가 발생하면 복사연산을 중단하게 되고 파일의 복사본은 원본보다 짧아질수 있다.

또한 기록형파일체계는 보통 모든 블로크를 기록에 저장하지 않는다. 각 파일체계는 두 종류 즉 메타자료와 일반자료를 저장하는 블로크로 구성되여있다. Ext2와 Ext3에는 초블로크, 그룹블로크서술자, i마디, 간접주소지정에 사용하는 블로크(간접블로크), 자료비트배치표블로크, i마디비트배치표블로크라는 모두 여섯종류의 조종자료가 있다. 다른 파일체계는 다른 조종자료를 사용한다.

ReiserFS, SGI의 XFS, IBM의 JFS와 같은 대부분의 기록형파일체계는 조종자료를 변경하는 연산들만 일지를 기록한다. 사실 조종자료의 로그레코드만 있으면 디스크의 파일체계자료구조의 일관성을 복구하는데 충분하다. 그러나 파일자료의 블로크에 대한 연산은 일지를 기록하지 않으므로 체계장애시 파일의 내용이 파괴되는것을 막아줄 방법은 없다.

그러나 Ext3파일체계는 파일체계조종자료와 파일의 자료블로크를 변경하는 모든 연산에 대해 일지를 기록하도록 설정할수 있다. 모든 종류의 쓰기연산에 대해 일지를 남기는것은 심각한 성능저하를 나타내기때문에 Ext3는 체계관리자가 무엇에 대해 일지를 남길지를 결정하도록 하고있다. 즉 다음과 같은 세 종류의 기록형방식을 제공한다.

- 기록(Journal)

모든 자료와 조종자료변경에 대해 기록일지를 남긴다. 이 방식은 매 파일에 대한 갱신을 잃을 가능성을 최소화하지만 추가적인 디스크접근을 많이 요구한다. 례를 들어 새로운 파일을 생성할 때 이 파일의 모든 자료블로크를 로그레코드에 복사해야 한다. 이 방식은 Ext3 기록형방식중에서 가장 안전하고 가장 느린 방식이다.

- 순차적(Ordered)

파일체계조종자료에 대해서만 기록을 남긴다. 그러나 Ext3파일체계는 조종자료와 이에 관련된 자료블로크를 그룹화해서 자료블로크들을 조종자료보다 먼저 디스크에 기록하도록 한다. 이렇게 함으로써 파일내부의 자료가 파괴되는 가능성을 최소화한다. 례를 들어 파일뒤에 추가하는 모든 쓰기접근은 기록기능이 완벽하게 보호하는 것을 보장한다. 이 방식은 Ext3의 기본기록방식이다.

- 되돌이쓰기(Writeback)

파일체계조종자료에 대한 변경만 일지를 남긴다. 이 방법은 다른 기록형파일체계에서도 사용하며 가장 빠른 방식이다.

Ext3파일체계의 기록형방식은 mount명령에 항목으로 지정할수 있다. 례를 들어 /dev/sda2구획에 저장된 Ext3파일체계를 /jdisk탑재지점에 되돌이쓰기방식으로 탑재하기 위해서 체계관리자는 다음과 같이 입력한다.

```
# mount    t ext3    o data=writeback /dev/sda2 /jdisk
```

8. 기록형블로크장치계층

Ext3기록기능은 보통 파일체계의 뿌리등록부에 .journal이라는 숨은 파일에 저장된다.

Ext3파일체계는 기록기능을 스스로 관리하지 않는다. 대신에 기록형블로크장치(JBD: Journaling Block Device)라는 일반핵심부계층을 사용한다. 지금은 Ext3만 JBD계층을 사용하지만 나중에는 다른 파일체계가 사용할수도 있다.

JBD계층은 좀 복잡한 쏘프트웨어이다. Ext3파일체계는 이후연산이 체계장애인 경우에도 디스크자료를 파괴하지 않도록 하기 위해 JBD루틴을 호출한다. 그러나 JBD는 보통 Ext3파일체계가 실행한 변화를 디스크에 일지로 남기려고 동일한 디스크를 사용하며 따라서 Ext3와 마찬가지로 체계장애에 취약하다. 다시 말하면 JBD는 기록을 깨뜨릴수도 있는 체계장애로부터 자기자신도 보호를 해야 한다. 따라서 Ext3와 JBD사이의 호상작용은 다음과 같은 기본적인 단위를 바탕으로 한다.

- 로그레코드(Log record)

기록형파일체계의 디스크블로크 하나에 대한 갱신 한번을 나타낸다.

- 원자적연산조종(Atomic operation handle)

파일체계에 대한 고수준변경하나를 나타내는 로그레코드를 포함한다. 파일체계를 변경하는 매 체계호출은 원자적작업조종 하나를 생성한다.

취급(Transaction)

여러 원자적작업조종로서 이것들의 로그레코드들은 e2fsck에 대해 동시에 유효하다고 표시된다.

1) 로그레코드

로그레코드는 파일체계가 실행하는 저수준연산하나를 나타낸다. 어떤 기록형파일체계에서는 로그레코드는 저수준연산이 변경한 정확한 바이트범위자료와 파일체계에서의 시작위치로 구성되여있다. JBD계층에서 로그레코드는 저수준연산이 변경한 완충기 전체로 구성되여있다. 이 접근방법은 기록공간을 랑비할수 있지만(례를 들면 저수준연산이 비트배치표의 한 비트를 변경했을 때) JBD계층이 완충기, 완충기머리부와 직접 작업할수 있으므로 훨씬 빠르게 동작한다. 따라서 로그레코드는 기록안에서 자료(또는 조종자료)의 블로크로 표현한다. 그러나 각 블로크에는 journal_block_tag_t형태인 작은 태그가 붙어있어서 파일체계에서 블로크의 론리적블로크번호와 몇가지 상태기발을 저장한다.

후에 완충기가 로그레코드에 속하거나 완충기가 대응하는 조종자료블로크보다 먼저 디스크에 흘리기되여야 하는 자료블로크인 경우(《순차적(ordered)》기록방식인 경우), JBD가 완충기를 처리하면서 핵심부는 journal_head자료구조를 완충기머리부에 붙인다. 이 경우에 완충기머리부의 b_private마당에 journal_head자료구조의 주소를 저장하고 journal_head기발을 설정한다.

2) 원자적연산조종

파일체계를 변경하는 체계호출은 보통 디스크자료구조를 처리하는 저수준연산의 련속으로 분리된다. 례를 들어 한 블로크의 자료를 정규파일의 뒤에 추가해달라는 사용자의 요청을 Ext3에서 처리해야 한다고 가정하자. 파일체계계층은 파일의 마지막블로크를 결정하고 파일체계에서 여유블로크를 찾고 적절한 블로크그룹의 자료블로크비트배치표를 갱신하고 새로운 블로크의 론리적번호를 파일의 i마디나 간접주소지정블로크에 저장하고 새로운 블로크의 내용을 기록하고 끝으로 i마디의 몇개의 마당을 갱신한다. 이렇게 사용자의 자료추가연산은 파일체계의 자료와 조종자료에 대한 여러 저수준연산으로 변환된다.

이제 추가연산의 중간에 체계장애가 발생하였다고 가정하자. 어떤 저수준 처리는 실행되였고 어떤것들은 실행되지 않았다. 물론 더 심각한 상황일수도 있다. 고수준연산이 파일 두개이상에 영향을 주고있을수도 있다.(례를 들면 한 등록부에서 다른 등록부로 파일을 옮기는 경우)

자료가 파괴되는것을 막기 위해 Ext3파일체계는 매 체계호출을 원자적으로 처리하도록 보장해야 한다. 원자적연산조종은 고수준연산 하나에 대응하는 디스크자료구조에 대한 저수준연산들의 집합이다. 체계장애에서 복구할 때 파일체계는 고수준연산전체가 적용되거나 아니면 저수준연산중에서 하나도 실행되지 않도록 보장한다.

원자적연산조종은 handle_t형태의 서술자로 나타낸다. 원자적연산을 시작하기 위해 Ext3파일체계는 journal_start() JBD함수를 호출한다. 이 함수는 필요하면 새로운 원자적연산조종을 할당하고 조종은 현재 취급(Transaction)에 추가한다. 디스크에 대한 저수준연산이 프로쎄스를 연기할수 있으므로 활성화된 조종의 주소는 프로쎄스서술자의 journal_info마당에 저장한다. Ext3파일체계는 원자적연산의 완료를 알리기 위해 journal_stop()함수를 호출한다.

3) 취급(Transaction)

효률을 높이기 위해서 JBD계층은 여러 원자적연산조종에 속하는 로그레코드를 취급으로 묶어서 관리한다. 그리고 한 조종에 속하는 모든 로그레코드는 반드시 같은 취급에 속해야 한다. 한 취급의 모든 로그레코드는 기록의 련속한 블로크에 저장한다. JBD계층은 각 취급전체를 하나로 처리한다. 례를 들면 어떤 취급에 속한 모든 로그레코드가 파일체계에 위임된 다음에만 취급이 사용한 블로크들을 회수한다. 취급은 생성되자마자 새로운 조종의 로그레코드를 받을수 있다. 취급은 다음 경우중 하나가 발생하면 새로운 조종을 받는것을 중단한다.

- 지정한 시간이 경과했을 때(보통 5s)
- 기록에 새로운 조종을 위한 여유블로크가 남아있지 않을 때

취급은 transaction_t형태의 서술자로 나타낸다. 가장 중요한 마당은 i_state로 취급의 현재상태를 나타낸다.

취급은 다음과 같은 상태일수 있다.

4) 완료됨(Complete)

취급에 포함된 모든 로그레코드를 물리적으로 기록하였다. 체계장애에서 복구할 때 e2fsck는 기록의 모든 완료된 취급을 검토하여 대응하는 블로크를 파일체계에 기록한다. 이 경우 i_state마당에는 T_FINISHED값을 저장한다.

5) 완료되지 않음(Incomplete)

취급의 로그레코드 한개이상을 아직 일지에 물리적으로 기록하지 않았다. 체계장애가 발생하면 일지에 저장된 취급의 영상은 최신이 아니기 쉽다. 따라서 체계장애에서 회복하는 과정에서 e2fsck는 기록에 있는 완료되지 않은 취급을 믿지 않고 무시하고 건너뛴다. 이 경우 i_state마당은 다음중 한 값을 저장한다.

T_RUNNING

여전히 새로운 원자적연산조종을 받는다.

T_LOCKED

새로운 원자적연산조종을 받지 않는다. 그러나 그중 일부는 아직 완료되지

않았다.

T_FLUSH

모든 원자적연산조종이 완료되였다. 그러나 로그레코드가 아직 기록되고있다.

T_COMMIT

원자적연산조종의 모든 로그레코드가 디스크에 기록되였으며 취급이 완료되였다고 기록에 표시한다.

어떤 순간에도 기록은 여러 취급을 가지고있을수 있다. 그중 단 하나만 T_RUNNING상태이고 이 취급이 Ext3파일체계가 요청한 새로운 원자적연산조종을 받는 활성화된 취급이다.

기록에 있는 취급중 어떤것은 로그레코드를 포함하고있는 완충기를 아직 기록하지 않아서 완료하지 않은 상태에 있을수 있다.

JBD계층이 검사하여 로그레코드가 나타내는 모든 완충기를 Ext3파일체계에 성공적으로 기록했음을 확인하면 완료한 취급은 기록에서 삭제한다. 따라서 완료한 취급 여러개와 완료하지 않은 기록을 최대 한개를 가질수 있다. 완료한 취급의 로그레코드는 기록되였지만 대응하는 완충기중 일부는 파일체계에 아직 기록되지 않았을수도 있다.

9. 기록기능의 동작

이제부터 기록기능이 어떻게 동작하는지 다음과 같은 례를 들어 보자.

Ext3파일체계계층이 어떤 정규파일의 일부 자료블로크를 기록하라는 요청을 받았다.

여기서는 Ext3파일체계계층과 JBD계층에 대해 각 행단위연산을 설명하지는 않는다. 이렇게 하려면 너무나 많은 내용을 다루어야 하므로 핵심적인 동작위주로 설명한다.

① write()체계호출의 봉사루틴은 Ext3정규파일에 련관된 파일객체의 write메쏘드를 시작하게 한다. Ext3의 경우 이 메쏘드는 generic_file_write()함수로 실현한다.

② generic_file_wrtie()함수는 쓰기연산에 관련된 자료의 각 페지에 대해 한번씩 address_space객체의 prepare_write메쏘드를 여러번 호출한다. Ext3의 경우 이 메쏘드는 ext3_prepare_write() 함수로 실현한다.

③ ext3_prepare_write()함수는 journal_start() JBD함수를호출하여 새로운 원자적연산을 시작한다. 운전은 활성취급에 추가된다. 원자적연산조종은 journal_start()함수를 처음 호출할 때 생성되고 이후 호출은 프로쎄스서술자의 journal_info마당이 설정되여있음을 확인하고 참조한 조종을 사용한다.

④ ext3_prepare_write()함수는 block_prepare_write()함수를 호출한다. 변수로서 ext3_get_block()함수의 주소를 전달한다. block_prepare_write()함수는 완충기와 파일의 페지의 완충기머리부를 관리한다.

⑤ 핵심부가 Ext3파일체계에서 어떤 블로크의 론리블로크번호를 알아내야 할 때 ext3_get_block()함수를 호출한다. 이 함수는 ext2_get_block()와 류사하다. 핵심적

인 차이는 Ext3파일체계에서 JBD계층의 함수를 호출하여 저수준연산의 일지가 남도록 보장한다는 점이다.

- 파일체계의 조종자료블로크에 대해 저수준의 쓰기연산을 요구하기 전에 함수는 journal_get_write_access()를 호출한다. 이 함수는 조종자료완충기를 활성취급의 목록에 추가한다. 그러나 조종자료가 기록의 오래되고 완료되지 않은 취급에 포함되여있는지 검사하여 이 경우 완충기를 복제하여 이전 취급은 이전 내용으로 위임되도록 해야 한다.
- 조종자료블로크를 포함하는 완충기를 갱신한 다음 Ext3파일체계는 journal_dirty_metadata()를 호출하여 조종자료완충기를 활성취급의 적절한 불결한 목록으로 옮기고 연산에 대해 일지를 기록한다.

JBD계층이 처리하는 조종자료완충기는 i마디완충기의 불결한 목록에 포함되지 않으며 따라서 일반디스크캐쉬처리기법에서는 디스크에 기록되지 않는다.

⑥ Ext3파일체계를 《기록》방식으로 탑재하였다면 ext3_prepare_write()는 쓰기연산이 값을 바꾼 모든 완충기에 대해 journal_get_write_access()를 호출한다.

⑦ 다시 generic_file_write()함수로 돌아와서 사용자방식의 주소공간에 저장된 자료로 페지를 갱신하고 address_space객체의 commit_write메쏘드를 호출한다. Ext3의 경우 이 메쏘드는 ext3_commit_write()함수로 실현한다.

⑧ Ext3파일체계를 《기록》방식으로 탑재하였다면 ext3_commit_write()는 페지의 모든 자료완충기(조종자료는 제외)에 대해 journal_dirty_metadata()를 호출한다. 이렇게 함으로써 완충기는 활성취급의 적절한 불결한 목록에 포함되고 소유자i마디의 불결한 목록에 들어가지 않는다. 그리고 대응하는 로그레코드를 기록한다.

⑨ Ext3파일체계를 《순차적》방식으로 탑재하였다면 ext3_commit_write()함수는 페지의 모든 자료완충기에 대해 journal_dirty_data()함수를 호출하여 완충기를 활성취급의 적절한 목록에 삽입한다. JBD계층은 취급의 조종자료 완충기가 기록되기전에 이 목록의 모든 완충기가 기록되는것을 보장해준다. 아무 로그레코드도 기록하지 않는다.

⑩ Ext3파일체계를 《 순차적 》 또는 《 되돌이쓰기 》 방식으로 탑재하였다면 ext3_commit_write()함수는 일반 generic_commit_write()함수를 호출한다. 이 함수는 자료완충기를 사용자i마디의 불결한 완충기목록에 삽입한다.

⑪ 끝으로 ext3_commit_write()는 journal_stop()을 호출하여 JBD계층에 원자적연산조종이 닫힌 사실을 알려준다.

⑫ write()체계호출의 봉사루틴이 여기서 완료된다. 그러나 JBD계층의 작업은 남아있다. 모든 취급의 로그레코드를 물리적으로 기록에 저장하면 취급은 완료상태가 된다. 그리고나면 journal_commit_transaction()을 실행한다.

⑬ Ext3파일체계를 《 순차적 》 방식으로 탑재하였다면 journal_commit_transaction()함수는 취급의 목록에 포함된 모든 자료완충기에 대해 입출력자료완충기

를 활성화하고 모든 자료전송이 완료될 때까지 기다린다.

⑭ journal_commit_transaction()함수는 취급에 포함된 모든 조종자료완충기에 대한 입출력자료전송을 활성화한다.(Ext3를 《기록》방식으로 탑재하였다면 모든 자료완충기를 포함한다.)

⑮ 핵심부는 주기적으로 기록에 있는 모든 완료된 취급에 대해 검사지적자를 활성화한다. 검사지적자는 journal_commit_transaction()이 시작한 입출력자료전송이 성공적으로 완료했는가를 검사한다. 그렇다면 취급을 기록에서 삭제할수 있다.

물론 기록에 있는 로그레코드는 체계장애가 발생할 때까지 어떤 실제적인 일도 하지는 않는다. 장애가 발생하면 e2fsck유틸리티프로그람이 파일체계에 저장된 기록을 탐색하고 완료된 취급의 로그레코드가 나타내는 모든 쓰기연산을 다시 순서짜기한다.

제 3 장. 프로쎄스관리

제 1 절. 프로쎄스

《프로쎄스(process)》는 모든 다중과제(multiprogramming)조작체계의 필수적인 개념이다. 프로쎄스는 일반적으로 실행상태에 있는 프로그람의 실체 (instance)로 정의한다. 따라서 사용자 16명이 동시에 vi프로그람을 실행하고있다면 (이것들이 똑같은 실행코드를 공유하더라도) 각각 다른 프로쎄스가 16개 존재하는 셈이다. Linux 원천코드에서는 프로쎄스를 가리켜 《과제(task)》 또는 《스레드(thread)》라고 부르기도 한다.

이 장에서는 먼저 프로쎄스의 정적인 특성을 살펴본 후 핵심부가 프로쎄스절환을 어떻게 하는가를 설명한다. 마지막 두 부분에서는 프로쎄스가 어떻게 만들어지고 없어지는가를 본다. 또한 Linux가 《1장 소개》에서 언급한 가벼운 프로쎄스(LWP: lightweight process)를 기반으로 하는 다중스레드응용프로그람을 어떻게 지원하는가도 설명한다.

1. 프로쎄스와 가벼운 프로쎄스, 스레드

《프로쎄스》라는 용어는 여러가지 다른 의미로 사용된다. 이 책에서는 일반적인 조작체계교과서에 나오는 정의(프로쎄스는 실행상태에 있는 프로그람의 실체이다.)를 따른다. 프로쎄스를 프로그람의 실행이 얼마나 진행되였는지를 완전하게 서술하는 자료구조의 집합이라고 생각해도 된다.

핵심부관점에서 보면 프로쎄스의 목적은 체계자원(CPU시간이나 기억기 등)을 할당받는 존재로서 동작하는것이다.

초기에 만들어진 프로쎄스는 부모와 거의 같다. 프로쎄스는 부모의 주소공간이 (론리적인) 복사본을 받아 부모와 똑같은 코드를 실행하며 프로쎄스를 만든 체계호출 다음에 있는 명령부터 시작한다. 부모와 자식은 프로그람코드(본문)가 들어있는 페지를 공유하지만 자료(탄창과 동적기억구역)는 서로 별개의 복사본을 가진다. 따라서 자식프로쎄스가 기억기의 내용을 바꾸더라도 부모프로쎄스에 보이지 않는다.(반대경우도 마찬가지이다.)

초기Unix핵심부는 이렇게 간단한 모형을 채택했지만 최근 Unix체계는 다르다. 요즘 응용프로그람자료구조의 대부분을 공유하면서 서로 독립적인 다수의 실행흐름으로 이루어진 사용자프로그람, 즉 《다중스레드응용프로그람(multithreaded application)》을 지원한다. 이런 체계에서 프로쎄스는 여러 《사용자스레드(user thread, 간단히 스레드라고도 한다.)》로 이루어지고 매 스레드는 프로쎄스의 실행흐름을 나타낸다. 요즘 표준서고인 《pthread(POSIX thread)》서고함수를 리용해서 대부분의 다중스레드응용프로그람을 작성한다.

이전 판본의 Linux핵심부에서는 다중스레드응용프로그람을 지원하지 않았다. 핵심부관점에서 보면 다중스레드응용프로그람은 그냥 보통 프로쎄스일뿐이였다. POSIX호환

인 pthread서고를 사용해서 다중스레드응용프로그람의 여러 실행흐름을 완전히 사용자방식에서 만들고 조종하며 순서짜기하였다.

그렇지만 이런 식의 다중스레드응용프로그람실현은 아주 만족스럽지는 않다. 례를 들어 스레드 두개를 사용하는 장기프로그람을 생각해보자. 둘중 하나는 그라픽장기판을 조종하고 유희하는 사람이 말을 옮기길 기다리며 콤퓨터가 말을 움직이면 이것을 화면에 보여준다. 다른 스레드는 유희의 다음 수를 생각한다. 첫번째 스레드가 사람이 말을 옮기길 기다리는동안 두번째 스레드를 계속 실행해서 사람이 생각하는 시간을 활용해야 한다. 그런데 장기프로그람을 한 프로쎄스로 구현한다면 첫번째 스레드는 사용자의 반응을 기다리면서 블로크상태로 들어가는 체계호출을 실행할수 있다. 이것을 호출하면 두번째 스레드 역시 차단될것이다. 대신 첫번째 스레드는 프로쎄스가 실행상태로 남아있을수 있도록 차단을 안하게 하는 복잡한 기법을 사용해야 한다.

Linux는 다중스레드응용프로그람을 잘 지원하기 위해 《가벼운 프로쎄스(lightweight process)》를 사용한다. 기본적으로 두 가벼운 프로쎄스는 주소공간이나 열린파일 등 여러 자원을 공유할수 있다. 그리고 둘중 하나가 공유하는 자원에 어떤 변경을 가하면 다른 쪽도 바뀐 부분을 바로 알수 있다. 두 프로쎄스는 공유자원을 접근할때 서로 동기화를 해야 한다.

가벼운 프로쎄스를 사용할수 있다면 각 스레드를 가벼운 프로쎄스와 련계함으로써 다중스레드응용프로그람을 쉽게 실현할수 있다. 이렇게 스레드는 간단히 같은 기억기주소공간과 같은 열린파일 등을 공유함으로써 같은 응용프로그람자료구조에 접근할수 있다. 동시에 핵심부는 매 스레드를 서로 독립적으로 순서짜기할수 있기때문에 한 스레드가 잠든 상태라도 다른 스레드는 실행상태로 있을수 있다.

Linux의 가벼운 프로쎄스를 사용하는 POSIX호환 pthread서고의 례로는《Linux 스레드(LinuxThread)》와 최근 IBM 에서 발표한 《다음세대 POSIX스레드화패키지(NGPT, Next Generation Posix Threading Package)》가 있다.

2. 프로쎄스서술자

프로쎄스를 다루려면 핵심부는 매 프로쎄스가 무엇을 하고있는지 명확히 알아야 한다. 례를 들어 핵심부는 프로쎄스의 우선순위, 프로쎄스가 실행상태에 있는지 아니면 어떤 사건을 기다리며 차단상태에 있는지, 프로쎄스에 어떤 주소공간이 할당되여있는지, 어떤 파일을 다룰수 있는지 등을 알아야 한다. 이것이 프로쎄스서술자(process descriptor) 즉 한 프로쎄스와 관련된 모든 정보를 담고있는 task_struct자료구조의 역할이다. 프로쎄스서술자는 매우 많은 정보를 저장하고있기때문에 상당히 복잡하다. 여기에는 프로쎄스의 특성을 가지고있는 많은 마당외에도 다른 자료구조를 가리키는 지적자를 포함하는 또 다른 자료구조에 대한 지적자도 여러개 있다. 그림 3-1은 Linux의 프로쎄스서술자를 도식화해서 보여준다.

그림에서 오른편에 있는 자료구조 다섯개는 프로세스가 소유하는 특정자원을 가리킨다. 이 자원에 대해서는 후에 설명하기로 하고 이 절에서는 프로세스상태를 나타내는 마당과 부모/자식(parent/child)간의 관계를 나타내는 마당, 이렇게 두 종류 마당에만 초점을 맞춘다.

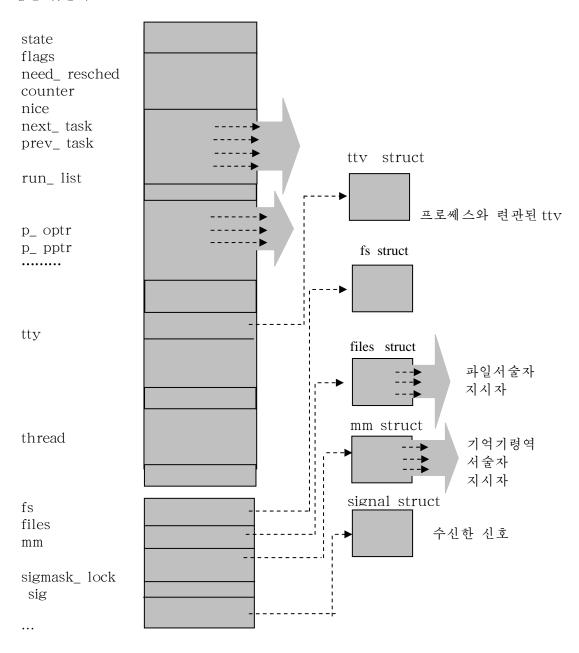



그림 3-1. 리눅스프로세스서술자

3. 프로쎄스상태

프로세스서술자의 상태마당은 이름그대로 프로쎄스에서 현재 무슨 일이 벌어지고있는가를 나타낸다. 이 마당은 기발의 배렬로 구성되며 매 기발은 가능한 프로쎄스상태를 나타낸다. 현재 Linux판본에서 매 상태는 호상배타적이다. 따라서 상태에 기발을 하나 설정하면 나머지 기발은 모두 지워진다. 다음은 가능한 프로쎄스상태목록이다.

TASK_RUNNING

프로쎄스가 CPU에서 실행중이거나 실행되기를 기다리는중이다.

TASK_INTERRUPTIBLE

프로쎄스가 어떤 조건이 맞아떨어지기를 기다리며 보류중(잠들어 있는중)이다. 프로쎄스를 깨울수 있는 조건으로는 하드웨어새치기가 발생하거나 프로쎄스가 기다리고있는 자원이 해제되거나 프로쎄스에 신호를 전달하는것이 있을수 있다.(프로쎄스는 깨여나면 TASK_RUNNING상태로 되돌아간다.)

TASK_UNINTERRUPTIBLE

우와 비슷하지만 잠들어있는 프로쎄스는 신호를 전달해도 프로쎄스상태가 바뀌지 않는다는 차이가 있다. 이 프로쎄스상태는 거의 사용하지 않는다. 그러나 프로쎄스가 정해진 사건이 발생하기를 기다리는 도중에 방해받으면 안되는 특수한 상황에서 효과적이다. 례를 들어 프로쎄스가 장치파일을 열 때 해당 장치구동프로그람이 자신이 다룰 하드웨어장치가 있는가를 조사하는 경우에 이 상태를 사용할수 있다. 장치구동프로그람은 조사를 완료할 때까지 방해받으면 안된다. 그렇지 않으면 하드웨어장치가 예측할수 없는 상태에 빠질수도 있다.

TASK_STOPPED

프로쎄스실행이 중단되였다. 프로쎄스는 SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU신호를 받으면 이 상태가 된다. 특정프로쎄스가 다른 프로쎄스를 감시하고있을 때(례를 들면 오유추적기가 ptrace()체계호출을 실행하여 다른 프로그람을 감시하는 경우) 신호는 프로쎄스를 TASK_STOPPED 상태로 만들수 있다.

TASK_ZOMBIE

프로쎄스실행은 완료했지만 부모프로쎄스가 wait()계렬체계호출(wait(), wait3(), wait4(), waitpid())을 호출하여 완료한 프로쎄스의 정보를 반환하지 않은 경우이다. 부모프로쎄스가 wait()계렬체계호출을 실행하기 전에는 완료한 프로쎄스의 프로쎄스서술자에 들어있는 자료를 부모프로쎄스가 필요로 할수 있기때문에 핵심부는 이 자료를 없애서는 안된다.(이 절 뒤에 있는 《프로쎄스제거》를 참고)

이외에도 프로쎄스상태에는 TASK_TRACED와 TASK_DEAD가 있다.

state마당값은 보통 다음과 같이 간단한 할당문을 사용하여 설정한다.

procdesc_ptr->state = TASK_RUNNING

핵심부에서는 set_task_state와 set_current_state마크로를 사용하기도 하는데 각각 지정한 프로쎄스와 현재 실행하는 프로쎄스의 프로쎄스상태를 설정한다. 더 나가서 이 마크로는 우의 대입연산이 콤파일러나 CPU조종단위에 의해 다른 명령와 섞이지 않게 한다. 명령순서가 섞이면 종종 왕청같은 결과가 나올수 있다.

4. 프로쎄스구별하기

독립적으로 순서짜기할수있는 모든 실행문맥은 자신만의 프로쎄스서술자를 가져야 한다. 따라서 핵심부자료구조의 웃부분을 서로 공유하는 가벼운 프로쎄스라도 자신만의 task_struct구조체를 가진다.

이렇게 프로쎄스와 프로쎄스서술자사이에는 엄격한 1:1대응관계가 있어서 32bit프로쎄스서술자주소를 리용해서 프로쎄스를 편리하게 구별할수 있다. 이런 주소를 《프로쎄스서술자지적자(process descriptor pointer)》라고 한다. 핵심부는 대부분 프로쎄스서술자지적자를 통해 프로쎄스를 참조한다.

반면에 모든 Unix계렬조작체계에서 사용자는 프로쎄스 ID(PID: process ID) 라는 수자로 프로쎄스를 구별한다. PID는 프로쎄스서술자의 pid마당에 저장하며 PID수자는 순차적으로 할당한다. 새로 만든 프로쎄스의 PID는 보통 바로 전에 만든 프로쎄스의 PID에 1을 더한 값이다. 그러나 16bit하드웨어작업기에서 개발한 고전Unix체계와 호환성을 유지하기 위해 Linux에서 사용할수 있는 최대 PID수자는 32767이다. 핵심부는 32768번째 프로쎄스를 만들 때 사용하지 않는 낮은 PID수자를 재활용하여 번호를 매기기 시작한다.

Linux는 체계에 있는 각각의 프로쎄스나 가벼운 프로쎄스에 서로 다른 PID를 부여한다.(뒤에서 보지만 다중처리기체계에서는 약간의 례외가 있다.) 이 방법은 체계에 있는 모든 실행흐름을 고유하게 구별할수 있어 매우 유연하다.

반면에 Unix프로그람작성자는 같은 그룹에 있는 스레드는 PID가 같을것이라고 생각한다. 례를 들어 PID를 하나만 지정하여 신호를 보내서 그룹에 들어있는 모든 스레드에 영향을 미칠수 있어야 한다. 사실 POSIX 1003.1c표준에서는 다중스레드응용프로그람의 모든 스레드는 PID가 같아야 한다고 서술하고있다.

표준과 호환성을 유지하기 위해 Linux2.6에서는 《스레드그룹(thread group)》이라는 개념을 도입하였다. 스레드그룹은 본질적으로 다중스레드응용프로그람의 스레드에 해당하는 가벼운 프로쎄스의 묶음이다. Task_struct 구조체의 thread_group마당에 있는 스레드그룹에 들어있는 모든 가벼운 프로쎄스의 서술자를 2중련결목록으로 묶는다. 스레드는 그 그룹에 있는 첫번째 가벼운 프로쎄스의 PID를 식별자로 공유하며 이것을 프로쎄스서술자의 rgid마당에 저장한다. getpid()체계호출은 current->pid가 아닌 current->tgid 를 반환한다. 따라서 다중스레드응용프로그람의 모든 스레드는 같은 식

별자를 공유한다. 일반프로쎄스나 스레드그룹에 들어가지 않는 가벼운 프로쎄스의 경우 gid마당은 pid마당과 값이 같다. 따라서 이러한 프로쎄스에는 getpid()체계호출이 평시처럼 동작한다.

PID를 가지고 실제프로쎄스서술자지적자를 효률적이고 뽑아내는 방법은 후에 본다. kill()을 비롯한 많은 체계호출은 대상이 되는 프로쎄스를 가리킬 때 PID를 사용하기때문에 PID를 효률적으로 검색하는것이 중요하다.

1) 프로쎄스서술자 다루기

프로쎄스는 수명이 짧아서 몇ms, 길어서 몇달에 이르는 동적인 존재이다. 따라서 핵심부는 동시에 많은 프로쎄스를 다룰수 있어야 하고 프로쎄스서술자를 항상 핵심부에 할당된 기억기령역에 저장하는것보다는 동적인 기억기에 저장하는것이 좋다. Linux는 8kB기억기령역 하나에 매 프로쎄스를 위한 서로 다른 두가지 자료구조를 저장한다. 하나는 프로쎄스서술자이고 다른 하나는 핵심부방식프로쎄스탄창이다.

프로쎄스는 핵심부방식에서 동작할 때 사용자방식에서 사용하는 탄창과 다른 핵심부자료토막에 있는 탄창을 사용한다고 설명하였다. 핵심부조종경로에서는 탄창을 조금밖에 사용하지 않기때문에 핵심부탄창은 몇천B정도만 필요하다. 따라서 8kB는 탄창과 프로쎄스서술자를 담는데 충분한 공간이다.

그림 3-2는 두 자료구조를 두 페지(8kB)크기의 기억기령역에 어떻게 저장하는가를 보여준다. 프로쎄스서술자는 기억기령역의 처음부터 시작하고 탄창은 끝에서 시작한다.

esp등록기는 탄창의 맨우의 위치를 가리키는 CPU의 탄창지적자(stack pointer)이다. Intel체계에서 탄창은 기억기령역의 끝에서 시작해서 령역의 시작쪽으로 커진다. 사용자방식에서 핵심부방식으로 절환한 후 프로쎄스의 핵심부탄창은 항상 비여있고 따라서 esp등록기는 기억기령역 바로 뒤에 있는 바이트를 가리킨다.

탄창에 자료를 기록하면 esp의 값은 감소한다. 프로쎄스서술자의 크기는 1000B가 채 안되기때문에 핵심부탄창은 7200B정도까지 늘어날수 있다.

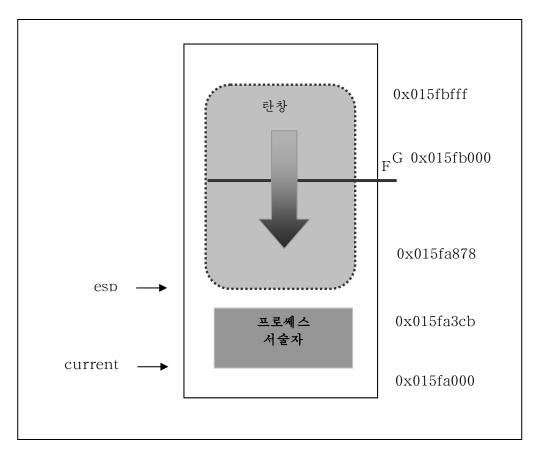



그림 3-2. 프로쎄스서술자와 프로쎄스핵심부탄창 저장하기

C언어에서는 다음과 같은 공용체(union)구조를 리용해서 프로쎄스서술자와 프로쎄스의 핵심부탄창을 간편하게 표현할수 있다.

```
union task_union {
        struct task_struct task;
        unsigned long stack[2048];
};
```

그림 3-2에서 프로쎄스서술자는 0x015fa000주소부터 들어있고 탄창은 0x015fc000부터 시작한다. 현재 탄창의 맨 우를 가리키는 esp등록기의 값은 0x015fa878이다.

핵심부는 프로쎄스서술자와 핵심부탄창을 저장하는 8kB크기의 기억기령역을 할당하고 해제할 때 각각 alloc_struct와 free_task_struct마크로를 사용한다.

2) current마크로

앞에서 설명한것처럼 프로세스서술자와 핵심부방식탄창을 긴밀하게 련계지으면 효률성측면에서 리득을 볼수 있다. 핵심부는 esp등록기값을 리용하여 CPU에서 현재 실행중인 프로세스의 프로세스서술자지적자를 쉽게 얻을수 있다. 실제로 기억기령역의 크기는 8kB(2^{13}B)므로 핵심부는 esp등록기의 아래 13bit만 지우면 프로세스서술자의 시작주소를 얻을수 있다. current마크로가 바로 이 일을 하는데 이 마크로를 콤파일하면 다음과 같은 기호언어명령이 만들어진다.

```
movl $0xffffe000, %ecx
andl %esp, %ecx
movl %ecx, p
```

이 쉘명령을 실행하면 p는 이 명령을 실행한 CPU에서 실행중인 프로세스의 프로세스서술자의 지적자를 가지게 된다.

핵심부코드에서 current마크로를 프로세스서술자마당앞에 앞붙이처럼 붙여서 쓰는것을 종종 볼수 있다. 례를 들어 current->pid는 CPU에서 현재 실행중인 프로세스의 프로세스 ID를 반환한다.

프로세스서술자와 탄창을 함께 저장하는 방식의 또 다른 우점은 다중 처리기체계에서 나타난다. 앞에서 본것처럼 단지 탄창을 검사하는것만으로 각 하드웨어처리기의 정확한 현재프로세스를 알아낼수 있다. Linux 2.0에서는 핵심부탄창과 프로세스서술자를 함께 지정하지 않았다. 대신 실행중인 프로세스의 프로세스서술자를 가리키는 대역정적변수인 current를 사용해야 하였다. 다중처리기체계에서는 current변수를 각 CPU에 해당하는 요소(element)를 하나씩 담은 배렬로 만들어야 하였다.

3) 프로세스목록

핵심부는 주어진 류형(례를 들면 실행가능한 상태에 있는 모든 프로세스)에 맞는 프로세스를 효률적으로 찾을수 있도록 여러 프로세스목록(process list)을 만든다. 매 목록은 프로세스서술자를 가리키는 지적자로 이루어진다. 프로세스서술자의 자료구조안에는 목록지적자(즉 각 프로세스가 다음 프로세스를 가리키는데 사용하는 마당) 하나가 들어있다. C언어로 정의한 task_struct구조체를 보면 이 자료구조는 재귀적인 방법으로 복잡하게 얽힌것처럼 보일수도 있다. 그러나 개념자체는 자신의 다음요소를 가리키는 지적자를 담는 자료구조인 여느 목록보다 복잡하지 않다.

원형2중련결목록(circular doubly linked list, 그림 3-3 참고)은 존재하는 모든 프로세스서술자를 서로 련결하는데 앞으로 이것을 프로세스목록(process list)이라고 한다. 목록을 실현하기 위해서 매 프로세스서술자에 있는 prev_task와 next_task마당을 사용한다. 목록의 머리는 init_task서술자이다. 이것은 모든 프로세스의 조상으로 《프로세스0(process 0)》또는 《교환기억기 (swapper)》라고 부른다.(뒤에 나오는 《핵

심부스레드》참고) init_task의 prev_task마당은 목록에 마지막으로 삽입된 프로쎄스서술자를 가리킨다.

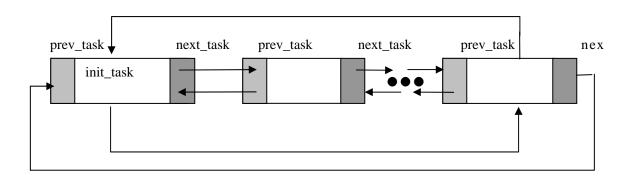



그림 3-3. 프로쎄스목록

프로쎄스목록에 프로쎄스서술자를 추가하거나 제거할 때에는 각각 SET_LINKS와 REMOVE_LINKS마크로를 사용한다. 이 마크로는 프로쎄스사이의 친족관계를 관계하면서 동작한다.(뒤에 나오는 《프로쎄스간 친족관계》참고)

또 다른 유용한 마크로로서 for_each_task가 있다. 이 마크로는 전체 프로쎄스목록을 탐색하며 다음과 같이 정의한다.

```
#define for_each_task(p) \
        for( p = &init_task; (p = p->next_task) |= &init_task;)
```

이 마크로는 순환을 조종하는 문장으로 핵심부프로그람작성자는 이 문장뒤에 순환에서 작업하는 코드를 넣는다. 여기서 init_task프로쎄스서술자가 어떻게 목록의 머리역할을 수행하는지 살펴보자. 이 마크로는 init_task를 지나 다음 작업으로 가면서 다시 init_task를 마주칠 때까지(목록이 원형으로 되여있는 덕분이다.) 순환을 반복한다.

4) 2중련결목록

프로쎄스목록은 특별한 2중련결목록이다. Linux핵심부는 다양한 핵심부자료구조를 저장하는 2중련결목록 수백개를 사용한다.

매 목록마다 목록을 초기화하고 항목을 추가하거나 삭제하고 목록을 탐색하는것과 같은 몇가지 기본적인 연산을 실현해야 한다. 이러한 연산을 서로 다른 목록마다 따로 만든다면 프로그람작성자의 노력과 기억기가 랑비되게 된다.

따라서 Linux핵심부에서는 list_head라는 자료구조를 정의하여 일반적으로 사용할 수 있는 2중련결목록을 실현한다. 이 자료구조의 next와 prev마당은 각각 2중련결목록의 다음과 이전에 있는 항목을 가리키는 지적자이다. list_head에 있는 지적자마당은 list_head자료구조를 포함하고있는 전체 자료구조의 주소가 아닌 다른 list_head마당의

주소를 저장한다는 사실이 중요하다.(그림 3-4 참고)

LIST_HEAD(list_name)마크로는 목록을 새로 만든다. 이 마크로는 list_head 형태의 list_name이라는 변수를 선언한다. 이 변수는 새로 만든 목록의 약속된 첫번째 요소이다.(init_task가 프로세스목록의 약속된 첫번째 요소인것과 비슷하다.)

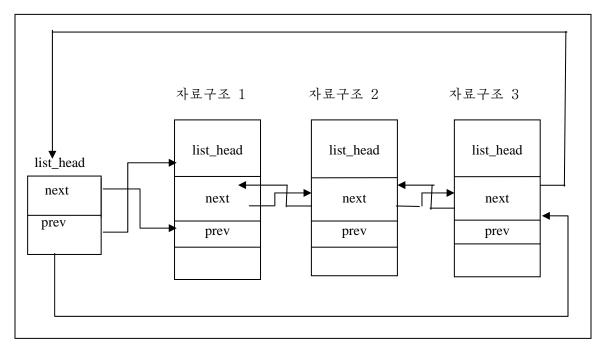



그림 3-4. list_head 자료구조를 리용한 2중련결목록

다음 목록에 소개하는것을 비롯하여 기본적인 연산을 실현하는 여러가지 함수와 마크로가 있다.

list_add(n,p)

p가 가리키는 요소 바로 다음에 n이 가리키는 요소를 삽입한다.(n을 목록의 맨 앞에 삽입하려면 p를 약속된 첫번째 요소의 주소로 지정하면 된다.)

list_add_tail(n,h)

약속된 첫번째 요소의 주소인 h로 지정한 목록의 맨 끝에 n이 가리키는 요소를 삽입한다.

list_del(p)

p가 가리키는 요소를 삭제한다.(목록의 약속된 첫번째 요소를 지정할 필요가 없다.)

list_empty(p)

약속된 첫번째 요소의 주소로 지정한 목록이 비여있는지 검사한다.

list_entry(p,t,f)

이름이 f이고 주소가 p인 list_head마당을 포함한 t형태자료구조의 주소를 반환한다.

list_for_each(p,h)

약속된 첫번째 요소의 주소 h로 지정한 목록에 있는 요소를 차례로 살펴본다.(프로쎄스목록에서 사용하는 for_each_task와 비슷하다.)

5) TASK_RUNNING프로쎄스목록

CPU에서 실행할 새로운 프로쎄스를 찾을 때 핵심부는 실행할수 있는 프로쎄스(즉 TASK_RUNNING상태에 있는 프로쎄스)만 고려해야 한다. 모든 프로쎄스목록을 검사하는것은 비효률적이므로 '실행렬(runqueue)'이라는 TASK_RUNNING프로쎄스의 원형2중련결목록(doubly linked circular list)을 도입하였다. 실행렬목록을 구현하기 위하여 프로쎄스서술자에는 list_head 형태인 run_list마당이 들어있다. 앞에 나온 프로쎄스목록과 마찬가지로 init_task프로쎄스서술자가 목록의 머리역할을 한다. nr_running변수는 실행가능한 모든 프로쎄스의 개수를 담는다.

add_to_runqueue()함수는 프로쎄스서술자를 목록의 맨앞에 삽입하고 del_from_runqueue()함수는 목록에서 프로쎄스서술자를 제거한다. move_first_runqueue()와 move_last_runqueue() 두 함수는 순서짜기를 할 때 사용하며 각각 프로쎄스서술자를 실행렬의 맨앞과 맨뒤로 옮긴다. task_on_runqueue()함수는 지정한 프로쎄스가 실행렬에 들어있는지 검사한다.

마지막으로 wake_up_process()함수는 프로쎄스를 실행가능하게 만들 때 사용한다. 이것은 프로쎄스상태를 TASK_RUNNING으로 만들며 add_to_runqueue()를 호출하여 프로쎄스를 실행렬에 넣는다. 또한 이 함수는 프로쎄스의 동적우선순위가 현재프로쎄스보다(또는 SMP체계라면 다른 CPU에서 현재 실행중인 어떠한 프로쎄스보다) 높은 경우 강제로 순서짜기를 호출한다.

6) pidhash표와 련쇄목록

핵심부에서 PID를 리용해서 이에 해당하는 프로쎄스서술자를 알아내야 하는 경우가 있다. 례를 들어 kill()체계호출을 처리하는 경우를 살펴보자. 프로쎄스 P1이 다른 프로쎄스 P2로 신호를 보내려고 P2의 PID를 변수로 지정하여 kill()체계호출을 실행한다. 그러면 핵심부는 PID에 해당하는 프로쎄스서술자의 지적자를 알아낸 다음 P2의 프로쎄스서술자에 있는 대기중인 신호를 기록하는 자료구조에 대한 지적자를 끌어낸다.

이때 프로쎄스목록을 차례로 검색해서 프로쎄스서술자에 있는 pid마당을 검사한다면 동작은 하겠지만 비효률적이다. 따라서 핵심부는 검색을 빠르게 하려고 입구점 PIDHASH_SZ개로 구성된 pidhash하쉬표를 도입하였다.(PIDHASH_SZ는 보통 1024

로 설정한다.) 이 표입구점은 프로세스서술자의 지적자를 포함한다. pid_hashfn마크로를 통해 PID를 표색인값으로 바꾼다.

```
#define pid_hashfn(x) (((x) >> 8) ^ (x)) & (PIDHASH_SZ - 1))
```

하쉬함수는 PID와 표색인값사이에 1:1대응을 보장하지 않는다. 서로 다른 두 PID가 같은 표색인값으로 하쉬되면 이것을 《충돌(colliding)》한다고 한다.

Linux는 충돌을 일으키는 PID를 다루기 위해 《련쇄(chaining)》기법을 사용한다. 매 표입구점은 충돌하는 프로세스서술자를 2중련결목록으로 엮는다. 이목록은 프로세스서술자에 있는 pidhash_nextd와 pidhash_prev 마당으로 실현한다. 그림 3-5는 목록 두개를 가진 pidhash표를 보여준다. PID가 119와 26799인 프로세스가 표의 200번째 요소로 하쉬되여 들어가며 PID 26800은 표의 217번째 요소로 들어간다.

어느 순간이든 체계에 있는 프로세스의 개수는 보통 32767(최대 PID값)보다 훨씬 작기때문에 련쇄목록(chained list)을 리용하여 하쉬하는것이 PID를 표색인값으로 그대로 변환하는것보다 낫다. 입구점 32768개로 구성된 표를 정의한다면 어느 순간이든 대부분의 입구점을 사용하지 않기때문에 저장공간을 랑비하게 된다. hash_pid()와 unhash_pid()는 pidhash표에 프로세스를 추가하거나 제거할 때 호출하는 함수이다. Find_task_by_pid()함수는 하쉬표를 검색하여 지정한 PID를 가진 프로세스의 프로세스서술자지적자를 반환한다.(프로세스를 찾지 못한다면 NULL지적자를 반환한다.)

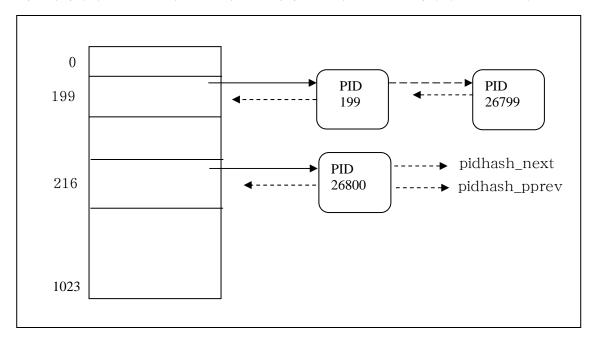



그림 3-5. 표와 련쇄목록

5. 프로쎄스간 친족관계

프로그람이 생성한 프로쎄스는 서로 부모/자식관계를 가진다. 한 프로쎄스가 여러 자식프로쎄스를 만들면 자식들은 형제(sibling)관계가 된다. 이런 관계를 나타내기 위해 프로쎄스서술자에는 여러 마당이 들어있다. 프로쎄스 0과 1은 핵심부가 만드는 프로쎄스이다. 뒤에서 보지만 프로쎄스 1(init)은 다른 모든 프로쎄스의 조상이다. 프로쎄스 P의 서술자에는 다음 마당이 들어있다.

p_opptr(원래 부모)

P를 만든 부모프로쎄스의 서술자는 부모프로쎄스가 더는 존재하지 않으면 프로쎄스1(init)의 프로쎄스서술자를 가리킨다. 따라서 쉘사용자가 배경프로쎄스(background process)를 시작하고 쉘을 빠져나가면 배경프로쎄스는 init의 자식이 된다.

p_pptr(부모)

P의 현재 부모(자식프로쎄스가 완료하면 신호를 받을 프로쎄스이다.)를 가리킨다. 이 값은 대개 p_opptr와 일치하지만 다른 프로쎄스가 ptrace()체계호출을 실행하여 P를 감시하게 해달라고 요청한 경우처럼 다룰수도 있다.

p_cptr(자식)

P의 가장 젊은 자식 즉 가장 최근에 만든 프로쎄스의 프로쎄스서술자를 가리킨다.

p_ysprt(동생)

P의 현재 부모가 P를 만들기 바로 전에 만든 프로쎄스의 프로쎄스서술자를 가리킨다.

p_osptr(형)

P의 현재 부모가 P를 만들기 바로 전에 만든 프로쎄스의 프로쎄스서술자를 가리킨다.

그림 3-6은 한 그룹의 프로쎄스간 부모와 형제관계를 보여준다. 프로쎄스 P0은 P1, P2, P3을, P3은 P4를 만들었다. P0은 p_pcptr부터 시작하여 형제를 가리키는 p_osptr지적자를 리용해서 모든 자식을 다 살펴볼수 있다.

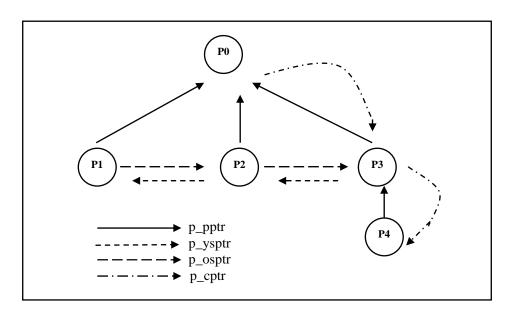



그림 3-6. 다섯프로쎄스사이의 친족관계

6. 프로쎄스조직화하기

실행렬(runqueue)목록은 TASK_RUNNING상태에 있는 모든 프로쎄스를 그룹으로 묶는다. 다른 상태에 있는 프로쎄스를 그룹으로 묶어야 할 때 각 상태에 따라 다르게 다루어야 하며 Linux는 다음중 하나를 고른다.

- TASK_STOPPED나 TASK_ZOMBIE상태에 있는 프로쎄스는 특별히 목록으로 련결하지 않는다. 정지상태나 좀비상태에 있는 프로쎄스는 PID나 특정프로쎄스에 속한 자식프로쎄스의 련결목록을 통해서만 접근하기때문에 이 두 상태에 있는 프로쎄스를 그룹으로 묶을 필요가 없다.
- TASK_INTERRUPTIBLE이나 TASK_UNINTERRUPTIBLE상태에 있는 프로쎄스는 시간에 따라 여러 부류로 나눌수 있다. 이 경우 해당 프로쎄스를 빨리 알아내려면 프로쎄스상태만으로는 부족해서 부가적인 프로쎄스목록 즉 대기렬(wait queue)을 도입해야 한다.

1) 대기렬

핵심부는 대기렬(wait queue)을 여러 용도로 사용한다. 특히 새치기처리와 프로쎄스동기화와 관련해서 많이 사용한다. 이 주제들은 뒤에서 이야기하기로 하고 여기서는 프로쎄스가 종종 그 어떤 사건(례를 들어 디스크작업이 끝나거나 어떤 체계자원을 사용할수 있게 되거나 정해진 시간이 지나는 등)이 일어나기를 기다려야 하는 경우에 대해서만 이야기한다. 대기렬은 이런 사건을 조건적으로 기다리는것을 실현한다. 프로쎄스는

어떤 사건이 일어나기를 기다릴 때 그에 맞는 대기렬에 들어가고 조종권을 포기한다. 따라서 대기렬은 잠든 상태에 있으며 기다리는 조건이 맞아떨어질 때 핵심부가 깨우는 프로세스의 집합이다.

대기렬은 프로세스서술자의 지적자를 포함하는 구조체의 2중련결목록으로 실현한다. 매 대기렬은 wait_queue_head_t형태자료구조인 《대기렬머리 (wait queue head)》로 식별한다.

```
struct __wait_queue_head {
        spinlock_t lock;
        struct list_head task_list;
};
typedef struct __wait_queue_head wait_queue_head_t;
```

주요핵심부함수뿐만아니라 새치기조종기에서도 대기렬을 사용하므로 이 2중련결목록을 동시에 접근하여 예측할수 없는 상황에 빠지는 일이 없도록 보호해야 한다. 대기렬머리에 있는 lock라는 스핀잠그기(spin lock)를 사용하여 동기화를 한다.

대기렬목록의 각 요소는 wait_queue_t형태이다.

대기렬목록의 각 요소는 어떤 사건이 일어나기를 기다리면서 잠들어있는 프로세스를 나타낸다. task마당에 있는 이 프로세스서술자의 주소가 들어간다. 그런데 프로세스를 깨울 때 대기렬에서 잠들어있는 《모든》 프로세스를 깨우는것이 항상 좋은것은 아니다.

례를 들어 독점적으로 접근할수 있는 어떤 자원을 사용할수 있기를 기다리는 프로세스가 두개이상 있다고 해보자. 이 경우 대기렬에 있는 프로세스중 하나만 깨우는것이 적합하다. 그러면 깨여난 프로세스가 자원을 사용하고 나머지 프로세스는 계속 잠들어있게 된다.(여러 프로세스가 깨여나서 그중 하나만 사용할수 있는 자원을 차지하려고 경쟁하고 결국 나머지 프로세스는 다시 잠든 상태로 돌아가는 문제를 피하기 위한것이다.)

따라서 잠든 프로세스는 두 종류로 나눌수 있다. 하나는 《배타적인 프로세스(exclusive process)》로서 (해당 대기렬요소의 flags마당값을 1로 설정하여 표시한다.) 핵심부가 선택적으로 깨우는 프로세스이다. 다른 하나는 《배타적이지 않는 프로세스(nonexclusive process)》로서(기발의 값을 0으로 설정하여 표시한다.) 기다리는 사건이 발생하면 핵심부가 항상 깨우는 프로세스이다. 배타적인 프로세스의 전형적인 례는 한번에 한 프로세스에만 할당할수 있는 자원을 기다리는 프로세스이다. 디스크작업이 끝나기를 기다리는 프로세스가 배타적이지 않는 프로세스의 례이다.

2) 대기렬처리

add_wait_queue()함수는 배타적이지 않은 프로세스를 대기렬목록의 맨앞에 삽입한다. add_wait_queue_exclusive() 함수는 배타적인 프로세스를 대기렬목록의 맨끝에 삽입한다. remove_wait_queue()함수는 프로세스를 대기렬목록에서 제거한다.

waitqueue_active()함수는 지정한 대기렬목록이 비여있는지 검사한다.

DECLARE_WAIT_QUEUE_HEAD(name)마크로를 리용하여 새로 대기렬을 정의할수 있다. 이 마크로는 name이라는 대기렬의 머리변수를 선언하고 초기화한다. init_waitqueue_head()함수는 미리 동적으로 할당한 대기렬머리변수를 초기화할 때 사용한다.

어떤 조건이 맞아떨어지기를 기다리는 프로쎄스는 다음 목록에 렬거한 함수중 하나를 호출할수 있다.

- sleep_on()함수는 현재프로쎄스에 다음작업을 한다.
  이 함수는 현재프로쎄스의 상태를 TASK_UNINTERRUPTIBLE로 설정하고 이것을 지정한 대기렬에 삽입한다. 그런 다음 순서짜기프로그람을 불러서 다른 프로쎄스의 실행을 재개한다. 잠들어 있던 프로쎄스가 깨여나면 순서짜기프로그람은 sleep_on()에서 실행을 재개하여 프로쎄스를 대기렬에서 제거한다.
- interruptibl_sleep_on()은 현재프로쎄스의 상태를 TASK_UNINTERRUPTIBLE이 아닌 TASK_INTERRUPTIBLE로 설정하여 신호를 수신해도 프로쎄스가 깨여날수 있게 한다는 점을 제외하고 sleep_on()과 똑같다.
- sleep_on_timeout()와 interruptible_sleep_on_timeout()함수는 앞에 나온 함수와 비슷하지만 시간이 얼마나 지나면 핵심부가 프로쎄스를 깨우겠는가도 지정할수 있다. 이것을 위해 이 함수는 schedule()대신 schedule_timeout()함수를 호출한다.
- Linux 2.6에서 도입한 wait_event와 wait_event_interruptible마크로는 지정한 조건이 참이 될 때까지 이것을 호출한 프로쎄스를 대기렬에서 잠들게 한다.
  례를 들어 wait_event_interruptible(wq, condititon)마크로는 다음과 같은 코드로 확장된다.

```
int __ret = 0;
if (!(condition)){
        DEFINE_WAIT(__wait);

        for (;;) {
                prepare_to_wait(&wq,&__wait,TASK_INTERRUPTIB
           LE);
        if (condition)
                break;
        if (!signal_pending(current)) {
                schedule();
```

```
                continue;
            }
            ret = -ERESTARTSYS;
            break;
        }
        finish_wait(&wq, &__wait);
        __ret;
```

이 마크로를 오래된 sleep_on()과 interruptible_sleep_on()대신 사용해야 한다. 그 리유는 이전 함수는 조건을 검사할수 있고 조건이 맞지 않을 때 프로쎄스를 원자적으로 재워서 잘 알려진 경쟁조건(race condition)의 원인이 되기때문이다.

앞에서 설명한 함수나 마크로를 통해 잠든 프로쎄스는 모두 배타적이지 않다. 핵심부는 대기렬에 배타적인 프로쎄스를 넣을 때에는 add_wait_queue_exclusive()를 직접 호출한다.

대기렬에 들어간 프로쎄스는 마크로 wake_up, wake_up_nr, wake_up_all, wake_up_sync, wake_up_sync_nr, wake_up_interruptible, wake_up_interruptible_nr, wake_up_interruptible_all, wake_up_interruptible_sync, wake_up_interruptible_sync_nr가운데서 하나를 사용하여 TASK_RUNNING상태로 들어갈수 있다.

이 마크로들은 include\linux\wait.h에 정의되여있다.

이 마크로 10개의 이름으로부터 매 마크로가 무슨 일을 하는지 알수 있다.

・모든 마크로는 TASK_INTERUPTIBLE상태로 잠들어있는 프로쎄스를 고려한다. 이름에 《 interruptible 》이란 문자렬을 포함하지 않은 마크로는 TASK_UNINTERRUPTIBLE상태에 있는 프로쎄스도 고려한다.

・모든 마크로는 요구한 상태(방금 전에 설명한)에 있는 모든 배타적이지 않는 프로쎄스를 깨운다.

・이름에 《nr》를 포함한 마크로는 지정한 개수만큼 요구한 상태에 있는 배타적인 프로쎄스를 깨운다. 이 개수는 마크로의 변수로 지정한다. 이름에 《all》을 포함한 마크로는 요구한 상태에 있는 모든 배타적인 프로쎄스를 깨운다. 마지막으로 이름에 《nr》나 《all》을 포함하지 않은 마크로는 요구한 상태에 있는 배타적인 프로쎄스를 단 하나만 깨운다.

・이름에 《sync》를 포함하지 않은 마크로는 깨여난 프로쎄스가 체계에서 현재 실행중인 프로쎄스보다 우선순위가 높은지 검사하여 필요하다면 schedule()함수를 호출한다. 이름에 《sync》를 포함한 마크로는 이런 검사를 하지 않는다.

례를 들어 wake_up마크로는 다음 코드조작으로 확장된다.

```
void wake_up(wait_queue_head_t *q)
```

```
{
        struct list_head *tmp;
        wait_queue_t *curr;

        list_for_each(tmp, &q->task_list) {
    curr = list_entry(tmp, wait_queue_t, task_list);
    wake_up_process(curr->task);
    if (curr->flags)
                    break;
        }
}
```

list_for_each마크로는 2중련결목록 q에 있는 모든 항목을 둘러본다.

list_for_each마크로는 다음과 같이 정의되여있다.

```
#define list_for_each(pos, head)
   for (pos = (head)->next, prefetch(pos->next); pos != (head);
        pos = pos->next, prefetch(pos->next))
```

list_entry마크로는 매 항목마다 해당 wait_queue_t변수의 주소를 계산한다. 변수의 task마당에는 프로쎄스서술자에 대한 지적자가 들어있어 wake_up_ process()를 호출할 때 이것을 전달한다. 깨여난 프로쎄스가 배타적인 프로쎄스라면 순환을 끝마친다. 배타적이지 않는 프로쎄스는 항상 2중련결목록의 앞쪽에 있고 배타적인 프로쎄스는 뒤쪽에 있기때문에 이 함수는 항상 배타적이지 않는 프로쎄스를 깨운 후 배타적인 프로쎄스가 있다면 이것을 하나만 깨운다. 이렇게 깨여난 프로쎄스를 대기렬에서 제거하지 않는데 주의해야 한다. 프로쎄스가 깨여날 때 여전히 기다리는 조건이 맞지 않을수 있고 그렇다면 프로쎄스는 같은 대기렬에서 다시 기다릴수도 있다.

7. 프로쎄스자원제한

각 프로쎄스마다 프로쎄스가 사용할수 있는 자원의 량을 지정하는 여러 자원제한(resource limit)이 있다. 이런 제한을 통해 사용자가 자원(CPU와 디스크공간 등)을 과도하게 사용하는것을 막을수 있다. Linux는 다음과 같은 자원제한을 지원한다.

RLIMIT_AS

프로쎄스주소공간의 최대크기(B단위). 핵심부는 프로쎄스가 malloc()나 자신의 주소공간을 늘이는 관련함수를 호출할 때 이 값을 검사한다.

RLIMIT_CORE

최대핵심쏟기(core dump)파일크기(B단위). 핵심부는 프로쎄스가 어떤 문제로 죽을 때 프로쎄스의 현재등록부에 core파일을 만들기 전에 이 값을 검사한다. 제한

값이 0이라면 핵심부는 이 파일을 만들지 않는다.

RLIMIT_CPU

프로쎄스가 사용할수 있는 최대 CPU시간(s단위). 프로쎄스가 이 제한시간을 넘어 동작하면 핵심부는 SIGXCPU신호를 보내고 그래도 프로쎄스가 완료하지 않으면 SIGKILL신호를 보낸다.

RLIMIT_DATA

최대동적기억구역크기(B단위). 핵심부는 프로쎄스의 동적기억 구역크기를 늘이기 전에 이 값을 검사한다.

RLIMIT_FSIZE

사용할수 있는 최대파일크기(B단위). 프로쎄스가 이 값보다 큰 파일을 만들려고 하면 핵심부는 SIGXFSZ신호를 보낸다.

RLIMIT_LOCKS

최대파일잠그기개수. 핵심부는 프로쎄스가 파일에 열쇠를 걸려고 할 때 이 값을 검사한다.

RLIMIT_MEMLOCK

교환할수 없는 기억기의 최대크기(B단위). 핵심부는 프로쎄스가 mlock()나 mlockall()체계호출을 리용하여 기억기에서 페지들을 잠그기(lock)하려고 할 때 이 값을 검사한다.

RLIMIT_NOFILE

최대로 열수 있는 파일서술자의 수. 핵심부는 새로 파일을 열거나 파일서술자를 복사할 때 이 값을 검사한다.

RLIMIT_NPROC

사용자가 소유할수 있는 최대 프로쎄스개수(뒤에 나오는 《clone(), fork(), vfork()체계호출》 참고)

RLIMIT_RSS

프로쎄스가 소유할수 있는 최대페지틀수. 핵심부는 프로쎄스가 mallock()나 자신의 주소공간을 늘이는 관련함수를 호출할 때 이 값을 검사한다.

RLIMIT_STACK

최대탄창크기(B단위) 핵심부는 프로쎄스의 사용자방식탄창크기를 늘이기전에 이 값을 검사한다.

자원제한은 프로쎄스서술자의 rlim마당에 저장한다. 이 마당은 각 자원제한마다 하나씩 있는 rlimit구조체의 배렬이다.

```
struct rlimit {
        unsigned long    rlim_cur;
        unsigned long    rlim_max;
```

};

rlim_cur마당은 자원의 현재사용제한을 나타낸다. 레를 들어 current->rlim[RLIMIT_CPU].rlim_cur는 실행중인 프로쎄스의 현재 CPU시간제한이다.

rlim_max마당은 자원제한에 사용할수 있는 최대값이다. 사용자는 getrlimit()와 setrlimit()체계호출을 리용하여 자원의 rlim_cur제한을 rlim_max까지 늘일수 있다. 그러나 초사용자(superuser) 좀 더 정확히 말하면 CAP_SYS_REWOURCE특성(capability)이 있는 사용자만이 rlim_max마당을 늘이거나 rlim_cur마당을 해당하는 rlim_max마당보다 크게 설정할수 있다.

대부분의 자원제한에는 보통 해당 자원에 제한이 없음을 의미하는 RLIMIT_INFINITY(0x7fffffff)값이 들어있다.(물론 핵심부설계에 따른 제한이나 사용가능한 RAM이나 디스크공간 등의 제한에 따른 실제 제한이 있다.) 그러나 체계관리자가 몇개의 자원을 엄격히 제한할수도 있다. 사용자가 체계에 가입할 때마다 핵심부는 setrlimit()를 호출하는 초사용자소유의 프로쎄스를 만들어 특정자원에 대한 rlim_max와 rlim_cur마당값을 줄일수 있다. 이 프로쎄스는 후에 등록가입쉘을 실행하고 사용자소유의 프로쎄스로 바뀐다. 사용자가 새로 만든 프로쎄스는 부모로부터 rlim배렬의 내용을 상속받으므로 사용자는 체계가 지정한 제한을 넘어설수 없다.

8. 프로쎄스절환

프로쎄스실행을 조종하기 위하여 핵심부는 CPU에서 실행중인 프로쎄스의 실행을 멈추고 이전에 멈춘 다른 프로쎄스의 실행을 재개할수 있어야 한다. 이러한 동작을 《프로쎄스절환(process switch)》또는 《작업절환(task switch)》, 《문맥절환(context switch)》이라고 한다.

1) 하드웨어문맥

매 프로쎄스는 자신만의 주소공간이 있지만 반면에 모든 프로쎄스는 CPU등록기들을 공유한다. 따라서 핵심부는 프로쎄스실행을 재개하기 전에 이 등록기들을 이전에 프로쎄스를 보류할 당시의 값으로 복구해야 한다.

이렇게 CPU에서 프로쎄스실행을 재개하기 전에 등록기로 다시 복구해야 하는 하드웨어집합을 《하드웨어문맥(hardware context)》이라고 한다. 하드웨어문맥은 프로쎄스절환에 필요한 모든 정보를 포함하는 프로쎄스 《실행문맥 (execution context)》의 일부이다. Linux에서는 프로쎄스하드웨어문맥의 일부를 프로쎄스서술자에 저장하고 일부는 핵심부방식의 탄창에 저장한다.

앞으로 국부변수 prev는 교체되여 나갈 프로쎄스의 프로쎄스서술자, next는 이것을 실행할 프로쎄스서술자를 가리킨다고 가정한다. 그렇다면 《프로쎄스절환(process switch)》은 prev의 하드웨어문맥을 저장한 뒤 next의 하드웨어문맥을 교체하는 작업이라고 할수 있다. 프로쎄스절환은 매우 자주 일어나므로 하드웨어문맥을 저장하고 복사

하는데 걸리는 시간을 최소화하는 일은 매우 중요하다.

이전 판본의 Linux는 Intel방식에서 제공하는 하드웨어지원을 활용하여 far jmp명령을 통해서 프로쎄스를 next프로쎄스의 작업상태토막서술자의 선택자로 절환하였다. 이 명령을 실행하면 CPU는 자동적으로 이전 하드웨어문맥을 저장하고 새로운 문맥을 적재한다. 그러나 다음과 같은 리유때문에 Linux 2.6에서는 프로쎄스절환을 쏘프트웨어적으로 처리한다.

・일련의 mov명령을 리용해서 프로쎄스를 절환하면 복구하는 자료가 옳바른지 더 확실히 통제할수 있다. 특히 토막등록기값을 검사할수 있다. 이런 종류의 검사는 간단히 far jmp명령을 사용하는 경우에는 불가능하다.

・이전 접근방법을 사용할 때 걸리는 시간과 새로운 방법을 사용할 때 걸리는 시간은 거의 같다. 그러나 현재프로쎄스절환코드는 앞으로 개선할수 있지만 하드웨어문맥으로 절환을 최적화하는것은 불가능하다.

프로쎄스절환은 핵심부방식에서만 일어난다. 프로쎄스가 사용자방식에서 사용하던 모든 등록기의 내용은 프로쎄스를 절환하기 전에 저장된다. 여기에는 사용자방식의 탄창지적자주소를 나타내는 ss와 esp의 내용도 포함된다.

2) 작업상태토막

80x86구조에는 하드웨어문맥을 저장하기 위한 《작업상태토막(TSS, Task State Segment)》이라는 특별한 토막이 있다. 비록 Linux는 하드웨어문맥절환을 사용하지 않지만 체계에 있는 각 CPU마다 별개의 TSS를 가지도록 하고있다. 이것은 다음 두가지 리유때문이다.

・80x86 CPU는 사용자방식에서 핵심부방식으로 절환할 때 TSS에서 핵심부방식의 탄창의 주소를 가져온다.

・사용자방식의 프로쎄스가 in이나 out명령을 사용하여 입출력포구에 접근하면 CPU는 프로쎄스가 해당 포구에 접근할수 있는지 확인하려고 TSS에 들어있는 입출력권한비트배치표(I/O permission bitmap)에 접근할수도 있다.

좀더 정확히 말하면 프로쎄스가 사용자방식에서 in이나 out 입출력명령을 사용하면 조종기구는 다음과 같은 일을 수행한다.

① eflags등록기의 IOPL마당을 검사한다. 이 마당이 3이면 조종기구는 입출력명령을 실행하고 그렇지 않으면 다음 검사를 한다.

② tr등록기에 접근하여 현재 TSS를 알아내고 이것을 통해 옳바른 입출력 권한비트배치표를 알아낸다.

③ 입출력권한비트배치표에서 입출력명령에 지정한 입출력포구에 해당하는 비트를 검사한다. 이 비트가 0이면 명령을 계속 실행하고 1인 경우 '일반보호오유(general protection error)'례외를 발생시킨다.

tss_struct구조체는 TSS의 형태를 정의한다. init_tss배렬은 체계에 있는 매 CPU마다 TSS를 하나씩 저장한다. 프로쎄스절환을 할 때마다 핵심부는 TSS의 몇개의 마당을 갱신하여 해당 CPU의 조종기구가 필요로 하는 정보를 안전하게 가져갈수 있게 한다.

매 TSS에는 8B크기인 자신만의 《작업상태토막서술자(TSSD: Task State Segment Descriptor)》가 있다. 이 서술자에는 TSS의 시작위치를 가리키는 32bit Base마당과 20bit Limit마당이 있다. TSSD의 S기발을 0으로 설정하여 TSS가 《체계토막(system segment)》임을 나타낸다.

type마당은 토막이 실제로 TSS임을 의미하는 9나 11값으로 설정된다. Intel의 원래 설계에 따르면 체계에 있는 각 프로쎄스는 자신만의 TSS를 참조해야 한다. type마당의 아래 두번째 비트를 《사용중인 비트(busy bit)》라고 한다. 프로쎄스가 CPU에서 실행중이면 이 비트는 1이고 그렇지 않으면 0이다. Linux설계에서는 각 CPU마다 TSS가 하나만 있기때문에 사용중인 비트는 항상 1이다.

Linux는 자신이 만든 TSSD를 GDT에 저장한다. GDT의 시작주소는 gdtr 등록기에 들어있다. 각 CPU의 tr등록기는 해당 TSS의 TSSD선택기를 포함한다. 이 등록기에는 프로그람작성할수 없는 두 마당(TSSD의 Base와 Limit마당)이 있다. 프로쎄스는 이 두 마당을 리용하여 GDT에서 TSS주소를 가져오지 않고도 TSS에 바로 접근할수 있다.

3) thread마당

프로쎄스절환을 할 때마다 교체되여 나가는 프로쎄스의 하드웨어문맥을 어딘가에 저장해야 한다. 이때 교체되여 나가는 프로쎄스가 실행을 재개할지 그리고 어떤 CPU가 이것을 다시 실행할지 알수 없기때문에 Intel의 원래 설계처럼 하드웨어문맥을 TSS에 저장할수는 없다.

따라서 각 프로쎄스서술자에는 thread_struct형태의 thread마당이 있다. 핵심부는 프로쎄스를 절환할 때마다 하드웨어문맥을 이곳에 저장한다.

후에 보지만 이 자료구조에는 범용등록기와 부동소수점등록기를 비롯하여 대부분의 CPU등록기가 들어간다.

4) 프로쎄스절환 수행하기

프로쎄스절환은 잘 정의한 단 한곳, 바로 schedule()함수에서만 일어날수 있다. 여기서는 핵심부가 어떻게 프로쎄스절환을 수행하는가에만 관심을 두겠다.

본질적으로 모든 프로쎄스절환은 두 단계로 이루어진다.

① 페지대역등록부를 교체하여 새로운 주소공간을 설치한다. 이 단계는 후에 다룬다.

② CPU등록기를 비롯하여 핵심부가 새로 프로쎄스를 실행하는데 필요한 모든 정보를 포함하는 핵심부방식탄창과 하드웨어문맥을 절환한다.

다시 한번 prev는 교체되여 나갈 프로쎄스의 서술자를, next는 새로 실행할 프로

쎄스의 서술자를 가리킨다고 가정한다. 후에 보지만 prev와 next는 schedule()함수의 국부변수이기도 하다.

switch_to마크로는 앞에서 언급한 프로쎄스절환의 두번째 단계를 수행한다. 이 마크로는 핵심부에서 가장 하드웨어의존적인 루틴중의 하나이며 이 마크로가 무슨 일을 하는지 알려면 상당한 노력을 기울여야 한다.

먼저 이 마크로는 prev와 next, last 이렇게 세 변수를 받는다. 다음은 schedule()에서 이 마크로를 실제로 호출하는 코드이다.

```
switch_to(prev, next, prev);
```

여기서 prev와 next의 역할을 쉽게 추측할수 있겠지만(국부변수 prev와 next 를 그대로 지정한것이다.) 세번째 변수 last는 무엇인가? 이것을 리해하는 핵심은 어떤 프로쎄스절환이든 두개가 아닌 세개의 프로쎄스가 관여한다는 사실이다.

핵심부가 프로쎄스 A를 절환하여 프로쎄스 B를 실행한다고 하자. schedule()함수에서 prev와 next는 각각 A와 B의 서술자를 가리킨다. switch_to마크로가 A를 멈추자마자 A의 실행흐름은 정지한다.

후에 핵심부가 A를 다시 실행한다고 하자. 이 경우 다른 프로쎄스 C(보통은 B와 다른 프로쎄스)를 교체해서 내보내야 할것이다. 이때 호출하는 switch_to마크로의 prev와 next는 각각 C와 A를 가리킨다. A가 실행을 재개하면 자신의 이전 핵심부방식탄창을 보게 되고 따라서 국부변수 prev와 next는 A와 B의 서술자를 가리킨다. 그러면 현재프로쎄스 A의 문맥에서 실행중인 핵심부는 C에 대한 참조를 잃게 된다.

switch_to마크로의 마지막변수는 C의 서술자의 주소를 prev변수에 다시 넣는 역할을 한다. 이 기구는 함수호출중의 등록기상태를 리용한다. 마크로를 시작할 때 첫번째 prev변수는 국부변수 prev의 내용으로 적재된 CPU등록기에 해당한다. 마크로가 끝날 때 똑같은 등록기의내용을 last변수에, 즉 국부변수 prev에 저장한다. 그런데 프로쎄스 절환을 할 때 CPU 등록기는 바뀌지 않기때문에 prev는 C의 서술자의 주소를 받게 된다.(순서짜기프로그람은 C를 다른 CPU에서 즉시 실행해야 하는지 검사한다.)

다음은 80x86극소형처리기에서 switch_to마크로가 수행하는 일에 관한 설명이다.

① prev와 next의 값을 각각 eax와 edx등록기에 저장한다.

```
movl prev, %eax
movl next, %edx
```

eax와 edx등록기는 각각 마크로의 prev와 next변수에 해당한다.

② prev를 ebx등록기로 복사한다. ebx는 마크로에서 last변수에 저장한다.

```
movl %eax, %ebx
```

③ esi, edi, ebp등록기의 내용을 prev핵심부방식탄창에 저장한다. 콤파일러는 switch_to마크로가 끝날 때 이 등록기의 값이 바뀌지 않고 남아있을것이라고 가정하기때문에 반드시 저장해야 한다.

```
pushl %esi
pushl %edi
pushl %ebp
```

④ esp값을 prev->thread.esp마당에 저장하여 이 마당이 prev핵심부방식탄창의 끝을 가리키게 한다.

```
movl %esp, 616(%eax)
```

여기서 피연산자 616(%eax)은 주소가 eax+616인 기억기쉘을 가리킨다.

⑤ esp를 next->thread.esp값으로 설정한다. 지금부터 핵심부는 next의 핵심부방식탄창에서 동작하므로 이 명령은 실질적으로 prev에서 next로 문맥절환을 수행한다. 프로쎄스서술자의 주소와 핵심부방식 탄창의 주소는 밀접한 관계가 있으므로 핵심부탄창을 바꾸는것은 현재프로쎄스를 바꾸는것을 의미한다.

```
movl 616(%edx), %esp
```

⑥ 표식 1(후에 자세히 설명한다)의 주소를 prev->thread.eip로 저장한다. 교체되여 나가는 프로쎄스가 후에 실행을 재개할 때 표식1이 있는 명령부터 실행한다.

```
movl $1f, 612(%eax)
```

⑦ next의 핵심부방식탄창에 next->thread.eip값을 넣는다. 이 값은 대부분 표식1이 있는곳의 주소이다.

```
push1 612(%eax)
```

⑧ C함수 __switch_to()로 이행한다.

```
jmp __switch_to
```

이 함수는 이전 프로쎄스와 새로운 프로쎄스를 나타내는 prev와 next를 변수로 받는다. 이 함수를 호출하는 방법은 일반적인 함수호출과 다르다. __switch_to()는 prev와 next변수를 다른 함수처럼 탄창에서 받지 않고 앞서 저장한 eax와 edx등록기에서 받는다. 핵심부는 이 함수가 변수를 등록기에서 받을수 있도록 gcc콤파일러의 비표준 C언어 확장중 하나인 __attribute__와 regparm을 사용한다.

이 함수는 switch_to()마크로에서 시작한 프로쎄스절환을 완료한다. 이 함수에는 등록기를 참조할 때 특별한 기호를 리용하여 좀 읽기 복잡하게 작성된 확장 inline기호언어코드가 들어있다.

unlazy_fpu()마크로코드를 실행하여 필요한 경우 FPU와 MMX, XMM 등록기내용을 저장한다.

후에 보지만 문맥절환을 할 때 next의 해당 등록기를 복구할 필요는 없다.

ㄱ. 현재 CPU에 대한 TSS의 esp0마당을 next->esp0으로 설정하여 후에 사용자방식에서 핵심부방식으로 특권준위가 바뀔 때 자동으로 이 주소가 esp등록기로 들어가게 한다.

```
init_tss[smp_processor_id()].esp0 = next->thread.esp0;
```

smp_processor_id()마크로는 현재 실행중인 CPU의 색인값을 반환한다.

ㄴ. fs와 gs토막등록기를 각각 prev->thread.fs와 prev->thread.gs로 저장한다. 해당 기호언어명령은 다음과 같다.

```
movl %fs, 620(%esi)
movl %gs, 624(%esi)
```

여기서 esi등록기는 prev->thread구조체를 가리킨다.

ㄷ. fs와 gs토막등록기를 각각 next->thread.fs와 next->thread.gs에 있는 값으로 설정한다. 이 단계는 바로 앞단계와 론리적으로 정반대이다. 해당 기호언어명령은 다음과 같다.

```
movl 12(%ebx), %fs
movl 17(%ebx), %gs
```

ebx등록기는 next->thread구조체를 가리킨다. 이 코드는 CPU가 잘못된 등록기값을 가지는 경우 례외를 발생시킬수 있기때문에 실제로는 훨씬 힘들다. 이런 가능성을 념두에 두고 이 코드는 《수선》접근방법을 채택하고있다.

ㄹ. next->thread.debugreg배렬에 있는 내용으로 오유등록기(debug register) 6개를 설정한다. 이것은 next가 보류되기 전에 오유등록기를 사용한 경우(즉 next->tss.debugreg[7]마당이 0이 아닌 경우)에만 수행한다. TSS에 값을 써넣어야만 이 등록기를 바꿀수 있다. 따라서 prev의 해당 등록기를 저장할 필요는 없다.

```
if (unlikely(next->debugreg[7])) {
        loaddebug(next, 0);
        loaddebug(next, 1);
        loaddebug(next, 2);
        loaddebug(next, 3);
        /* 4 번 5번은 제외 */
        loaddebug(next, 6);
        loaddebug(next, 7);
}
```

ㅁ. 필요하다면 TSS에 있는 입출력접근권한비트배치표를 갱신한다. next나 prev가운데서 하나라도 자신만의 입출력접근권한비트배치표가 있다면 이 작업을 해야 한다.

```
if (next->thread.ioperm) {
      memcpy(init_tss[smp_processor_id()].io_bitmap,
        next->thread.io_bitmap, 128);
```

```
            init_tss[smp_processor_id()].bitmap = 104;
        } else if (prev->thread.ioperm)
            init_tss[smp_processor_id()].bitmap = 0x8000;
```

프로세스에 맞게 바꾼 입출력접근권한비트배치표는 프로세스서술자의 thread.io_bitmap마당이 가리키는 완충기에 들어있다. 만약 next가 이런 비트배치표를 가지고있다면 이것을 TSS의 io_bitmap마당으로 복사한다. 그렇지 않으면 핵심부는 prev가 이런 비트배치표를 가지고있는가를 검사하여있다면 비트배치표를 무효화한다.

ㅂ. 완료한다. 다른 함수와 마찬가지로 __switch_to()도 ret 기호언어명령을 사용하여 완료한다. 이 명령을 실행하면 CPU는 eip프로그람계수기를 탄창에 들어있는 돌아갈 주소로 설정한다. 그런데 __switch_to()함수를 호출할 때 여기에 이행(jump)명령을 내렸다. 따라서 ret기호언어명령은 돌아갈 주소로 switch_to 마크로가 탄창에 저장한 표식1이 붙은 명령(바로 다음에 나온다)의 주소를 발견하게 된다. next를 처음으로 실행해서 이전에 보류한 사실이 없는 경우라면 ret_from_fork()함수의 시작주소를 가지게 된다.

⑨ esi, edi, ebp등록기를 복구하는 명령 몇개가 남았다. 이 세 명령중 첫번째 명령에 1이라는 표식이 붙는다.

```
1:
    popl %ebp
    popl %edi
    popl %esi
```

이 pop명령이 어떻게 prev프로세스의 핵심부탄창을 사용하는지 눈여겨보자. 순서짜기가 CPU에서 새로 시작할 프로세스로 prev를 선택하여 switch_to를 호출할 때 두번째 변수로 prev를 넘겨줄 경우 이 명령을 실행한다. 따라서 그 순간의 esp등록기는 prev의 핵심부방식탄창을 가리킨다.

⑩ ebx등록기의 내용(switch_to마크로의 last변수에 해당한다)을 국부변수 prev로 복사한다.

```
movl %ebx, prev
```

앞서 설명한것처럼 ebx등록기는 바로 전에 교체되여 나간 프로세스의 서술자를 가리킨다.

9. FPU와 MMX, XMM등록기저장

Intel80486처리기부터 계산을 담당하는 부동소수점계산기구(FPU)를 CPU에 통합하기 시작하였다. 특별히 제작한 비싼 소편에서 부동소수점계산을 하여왔기때문에 지금도 《수학보조처리기(mathematical coprocessor)》라는 이름을 사용한다. 부동소수점

계산기능은 이전 모형과 호환성을 유지하기 위해 《확장명령(escape instruction)》을 활용한다. 확장명령이란 0xd8에서 0xdf 범위에 있는 앞붙이바이트(prefix byte)를 사용하는 명령이다. 부동소수점 명령은 CPU에 있는 부동소수점등록기를 통해 동작하기때문에 프로쎄스가 확장명령을 사용하면 부동소수점등록기의 내용 역시 하드웨어문맥에 들어간다.

Intel은 후기 펜티움모형부터 극소형처리기에 다매체응용프로그람의 실행속도를 높이기 위한 《MMX명령(MMX instructions)》이라는 새로운 기호언어명령을 추가하였다. MMX명령은 FPU에 있는 부동소수점등록기를 사용한다. 이런 선택에는 프로그람작성자가 부동소수점명령과 MMX명령을 혼합해서 사용할수 없다는 명백한 부족점이 있다. 그러나 조작체계를 설계하는 사람의 립장에서는 FPU의 상태를 저장하는 작업절환코드를 MMX상태를 저장하는데도 그대로 사용할수 있으므로 이 새 명령을 무시할수 없다는 우점이 있다.

MMX명령은 프로쎄스내부에 단일명령다중자료(SIMD, single-instruction multiple-data) 관흐름(pipelind)을 도입하여 다매체응용프로그람의 실행속도를 높인다. 펜티움3부터는 이런 SIMD성능을 더 확장하여 Streaming SIMD확장(extension)의 략자인 《SSE확장》을 도입하였다. 이것은 128bit 크기등록기(XMM등록기) 8개에 있는 부동소수점값을 처리하는 기능을 추가한것이다. 이 등록기는 FPU와 MMX등록기와 겹치지 않기때문에 SSE와 FPU/MMX명령을 자유롭게 섞어서 사용할수 있다. 펜티움4모형에서는 SSE2확장이라는 다른 특징을 도입하였다. 이것은 기본적으로 더 높은 정밀도의 부동소수점값을 지원하는 SSE의 확장이다. SSE2는 SSE와 마찬가지로 같은 XMM등록기를 사용한다.

80x86극소형처리기는 FPU와 MMX, XMM등록기를 TSS에 자동으로 저장하지 않는다. 그러나 필요할 때에만 핵심부가 이 등록기를 저장할수 있도록 몇가지 하드웨어적인 지원을 한다. 이런 하드웨어적인 지원은 cr0 등록기의 TS(Task Switching)기발을 통해 이루어지며 이 기발은 다음 규칙을 따른다.

- 하드웨어문맥을 절환할 때마다 TS기발을 1로 설정한다.
- TS기발을 설정한 상태에서 확장명령이나 MMX, SSE, SSE2명령을 실행할 때마다 조종기구는 《장치를 사용할수 없음(Device not available)》이라는 례외를 발생시킨다.

TS기발은 핵심부가 정말 필요한 경우에만 FPU와 MMX, XMM등록기를 저장하고 복구하게 한다. 이것이 어떻게 동작하는가 알아보기 위해 프로쎄스 A가 수학보조처리기를 사용한다고 가정하자. 문맥절환이 일어날 때 핵심부는 TS기발을 1로 설정하고 부동소수점등록기를 프로쎄스 A의 TSS에 저장한다. 새로운 프로쎄스 B가 수학보조처리기를 사용하지 않으면 핵심부는 부동소수점등록기의 내용을 복구할 필요가 없다. 그러나 B가 확장명령이나 MMX명령을 실행하려고 하면 CPU는 장치를 사용할수 없다는 례외

를 발생시키고 해당 조종기는 프로쎄스 B의 TSS에 저장되여있는 값을 부동수소점등록기로 읽어들인다.

이제 FPU와 MMX, XMM등록기를 선택적으로 적재하기 위해 도입한 자료구조를 살펴보자. i387_union공용체형식의 thread.i387마당에 이것을 저장한다.

```
union i387_union {
        struct i387_fsave_struct fsave;
        struct i387_fxsave_struct fxsave;
        struct i387_soft_struct soft;
};
```

보다싶이 이 마당은 3가지 자료구조중 하나만을 저장할수 있다. i387_soft_struct형태는 수학보조처리기가 없는 CPU모형에서 사용한다. Linux는 쏘프트웨어적으로 수학보조처리기를 흉내내서 여전히 이전의 집적소자를 지원한다. i387_fsave_struct형태는 수학보조처리기와 선택적으로 MMX기구를 포함한 CPU모형에서 사용한다. 마지막으로 i387_fxsave_struct형태는 SSE와 SSE2확장기능을 포함한 CPU모형에서 사용한다.

프로쎄스서술자는 다음 두 기발을 추가로 포함한다.

- flags기발에 들어있는 PF_USEDFPU기발은 CPU에서 프로쎄스를 현재 실행할 때 FPU나 MMX, XMM등록기를 사용했는지 여부를 나타낸다.
- used_math마당. 이 마당은 thread.i387마당에 들어있는 값이 의미가 있는지 나타낸다. 다음 두 경우에 이 기발을 0으로 설정한다.(의미가 없다고 표시한다.)
  - 프로쎄스가 execve()체계호출을 실행하여 새 프로그람을 실행하기 시작할 때 절대로 이전 프로그람으로 조종권이 되돌아가지 않기때문에 현재 thread.i387에 저장된 자료는 다시 사용하지 않는다.
  - 사용자방식에서 실행하던 프로쎄스가 신호조종기함수를 실행하기 시작할 때 신호조종기는 프로그람실행흐름측면에서 보면 비동기적이기때문에 신호기조종기에 부동소수점등록기는 아무 의미가 없다. 핵심부는 신호기를 실행하기 전에 thread.i387에 부동소수점등록기를 저장하고 조종기가 끝난 후에 이것을 복구한다. 따라서 신호기조종기에서는 수학보조처리기를 사용할수 있으나 정상적인 프로그람실행과정에서 시작한 부동소수점계산을 계속할수는 없다.

앞에서 설명한것처럼 __switch_to()함수는 교체되여 나가는 프로쎄스의 프로쎄스서술자를 변수로 지정하여 unlazy_fpu마크로를 실행한다. 이 마크로는 prev의 PF_USEDFPU기발을 검사한다. 기발이 1이면 실행할 때 FPU나 MMX, SSE, SSE2명령을 사용한것이므로 하드웨어문맥을 저장해야 한다.

```
if (prev->flags & PF_USEDFPU)
        save_init_fpu(prev);
```

save_init_fpu()함수는 다음 작업을 수행한다.

· FPU등록기의 내용을 prev프로쎄스서술자에 저장하고 FPU를 다시 초기화한다. CPU가 SSE/SSE2확장을 사용하면 XMM등록기의 내용도 저장하고 SSE/SSE2기구를 다시 초기화한다. 강력한 기호언어명령 몇개를 사용해서 이 모든 일을 할수 있다.

CPU가 SSE/SSE2확장을 사용하면 다음과 같이 한다.

```
asm volatile ( "fxsave %0; fnclex"
        : "=m" (tsk->thread.i387.fxsave));
```

그렇지 않으면 다음과 같이 한다.

```
asm volatile ( "fnsave %0; fwait"
        : "=m" (tsk->thread.i387.fsave));
```

· prev의 PF_USEDFPU기발을 지운다.

```
prev->flags &= ~PF_USEDFPU;
```

· stt()마크로를 사용하여 cr0의 TS기발을 1로 설정한다. 이것은 실질적으로 다음과 같은 기호언어명령으로 확장한다.

```
movl %cr0, %eax
orl  $8, %eax
movl %eax, %cr0
```

프로쎄스가 실행을 재개하자마자 부동소수점등록기의 내용을 복구하지는 않는다. 그러나 unlazy_fpu()마크로에서 cr0의 TS기발을 1로 설정했기때문에 프로쎄스가 확장명령이나 MMX, SSE, SSE2명령을 처음으로 실행하려하면 조종기구는 《장치를 사용할수 없음》례외를 발생시키고 핵심부는(정확히 말해서 이 례외를 처리하는 례외조종기) math_state_restore()함수를 실행한다.

프로쎄스가 FPU나 MMX, SSE/SSE2명령을 실행하고있으므로 이 함수는 PF_USEDFPU기발을 1로 설정한다. 나가서 cr0의 TS기발을 0으로 설정하여 앞으로 프로쎄스가 FPU나 MMX,SSE/SSE2명령을 실행해도 《장치를 사용할수 없음》례외를 더는 발생시키지 않게 한다. thread.i387 마당에 저장한 자료가 유효하면 restore_fpu()함수는 부동소수점등록기에 적절한 값을 넣는다. 이를 위해 CPU가 SSE나 SSE2를 지원하는지 여부에 따라 fxrstor나 frstor기호언어명령을 사용한다. 반대로 thread.i387마당에 저장한 자료가 유효하지 않으면 FPU/MMX기구를 다시 초기화하고 모든 등록기를 지운다. SSE/SSE2기구를 다시 초기화하는것은 XMM등록기에 값을 적재하는것으로도 충분하다.

10. 프로쎄스생성

Unix조작체계는 사용자의 요구에 응해서 많은 프로쎄스를 생성한다. 례를 들어 쉘프로쎄스는 사용자가 명령을 입력할 때마다 쉘의 복사본을 실행하는 새로운 프로쎄스를

만든다.

전통적인 Unix체계는 모든 프로세스를 같은 방식으로 다루는데 부모프로세스가 보유한 자원을 복사하여 자식프로세스에 복사본을 주는것이다. 이런 방식은 부모프로세스의 모든 주소공간을 복사해야 하므로 프로세스생성이 매우 느리고 비효률적이다. 자식프로세스는 부모에서 상속받은 모든 자원을 읽거나 고칠 필요가 거의 없다. 많은 경우 곧바로 execve()를 호출하여 힘들여 복사한 주소공간을 지워버린다.

최근 Unix핵심부는 이 문제를 3가지 방식으로 해결한다.

- ≪쓰기복사≫기법은 부모와 자식이 모두 똑같은 물리적인 페지를 읽을수 있게 한다. 둘중 하나라도 페지에 쓰려고 하면 핵심부는 쓰려는 프로세스에 새로 물리적인 페지를 할당하여 내용을 복사해준다. 이 기법을 실현하는 방법은 후에 자세히 설명한다.
- 가벼운 프로세스(lightweight process)는 부모와 자식 모두 페지표(즉 전체 사용자방식주소공간)와 열린파일목록, 신호기배렬을 비롯하여 프로세스마다 할당하는 많은 핵심부자료구조를 공유한다.
- vfork()체계호출은 부모의 기억기주소공간을 공유하는 프로세스를 만든다. 부모프로세가 자식이 필요로 하는 자료를 덮어쓰지 않도록 자식프로세스가 완료하거나 새로운 프로그람을 실행하기 전까지 부모프로세스를 차단한다. vfork()체계호출은 후에 자세히 설명한다.

1) clone(), fork(), vfork() 체계호출

Linux에서는 clone()함수를 호출하여 가벼운 프로세스를 생성한다.

이 함수는 변수 4개를 받는다.

fn

새로운 프로세스로 실행할 함수를 지정한다. 이 함수에서 돌아오면 자식 프로세스는 완료한다. 이 함수는 자식프로세스의 완료값을 나타내는 정수를 반환한다.

arg

fn()함수에 전달할 자료를 가리킨다.

flags

잡다한 정보. 아래바이트는 자식프로세스가 완료할 때 부모프로세스에 전달할 신호번호를 나타낸다. 보통 이 신호는 SIGCHLD다. 나머지 3B는 복제와 관련한 기발로 부모와 자식프로세스사이에서 공유할 자원을 지정한다. 매 기발의 의미는 다음과 같다.

CLONE_VM

기억기서술자와 모든 페지표를 공유한다.

CLONE_FS

뿌리등록부와 현재 작업등록부를 나타내는 표와 《umask》라고 부르는 새

교육성 교육정보쎈터

로 만들어지는 파일의 초기접근권한의 비트마스크(bitmask)값을 공유한다.

CLONE_FILES

열린 파일을 나타내는 표를 공유한다.

CLONE_PARENT

새로 만들어지는 자식프로쎄스의 부모를(프로쎄스서술자에 있는 p_pptr와 p_opptr마당) 이 함수를 호출하는 프로쎄스의 부모로 설정한다.

CLONE_PID

PID를 공유한다.

CLONE_PTRACE

ptrace()체계호출을 리용하여 부모프로쎄스를 추가하고있다면 자식프로쎄스 역시 추적할수 있다.

CLONE_SIGHAND

신호조종기를 나타내는 표를 공유한다.

CLONE_THREAD

자식프로쎄스를 부모프로쎄스와 같은 스레드그룹에 넣고 자식프로쎄스의 tgid마당도 이에 따라 설정한다. 이 기발을 지정하면 CLONE_PAREANT 역시 설정된다.

CLONE_SIGNAL

CLONE_SIGHAND와 CLONE_THREAD를 동시에 지정한것과 같다. 따라서 다중스레드응용프로그람에 있는 모든 스레드에 신호를 보내는것이 가능하다.

CLONE_VFORK

vfork()체계호출에서 사용한다.

child_stack

자식프로쎄스의 esp등록기에 저장할 사용자방식탄창지적자를 가리킨다. 이 값이 0이면 핵심부는 현재 부모프로쎄스의 탄창지적자로 자식프로쎄스의 탄창지적자를 설정한다. 따라서 부모와 자식은 일시적으로 똑같은 사용자방식탄창을 공유한다. 그러나 《쓰기복사》기구때문에 두 프로쎄스중 하나라도 탄창의 내용을 바꾸려고 하자마자 별도의 복사본을 가지제 된다. 그러나 자식프로쎄스가 부모와 똑같은 주소공간을 공유하는 경우라면 이 변수는 null이 아닌 값이여야 한다.

clone()은 실제로는 C서고에 정의된 일종의 래퍼함수이며 프로그람작성자에게는 숨겨있는 Linux의 clone()체계호출을 사용한다. 이 체계호출은 flags와 child_stack 변수만 받는다. 새 프로쎄스는 항상 체계호출을 실행한 직후에 있는 명령부터 실행하기 시작한다. clone()함수는 체계호출에서 돌아오면 자신이 부모프로쎄스인지 자식프로쎄스인지를 검사하여 자식인 경우 fn()함수를 실행한다.

Linux는 clone()체계호출을 리용하여 전통적인 fork()체계호출을 실현한다. 이때

flags변수에 SIGCHLD신호와 함께 다른 clone기발은 지정하지 않으며 child_stack변수에 0을 지정한다.

Linux는 앞절에서 설명한 전통적인 vfork()체계호출 역시 clone()체계호출을 리용하여 실현한다. 첫번째 변수에 SIGCHLD신호와 함께 CLONE_VM 파 CLONE_VFORK기발을 지정하며 두번째 변수에 0을 지정한다.

clone()이나 fork(), vfork()체계호출을 실행하면 핵심부는 do_fork()함수를 호출하며 이 함수는 다음과 같은 단계를 수행한다.

① CLONE_PID기발을 지정한 경우 do_fork()함수는 부모프로쎄스의 PID 가 0인지 검사한다. 0이 아니라면 오유코드를 돌려준다. 《교환기(swapper)》프로쎄스만이 CLONE_PID를 설정할수 있다. 이것은 다중처리기체계를 초기화하는데 필요하다.

② alloc_task_struct()함수를 호출하여 새로운 프로쎄스의 프로쎄스서술자와 핵심부방식탄창을 담을 8kB크기의 union task_union기억기령역을 할당한다.

③ current지적자를 리용해 부모프로쎄스서술자를 앞에서 할당받은 기억기령역에 있는 새로운 프로쎄스서술자로 복사한다.

④ 사용자가 새 프로쎄스를 시작하는데 충분한 자원을 보유하고있는가를 확인하려고 몇가지 검사를 수행한다. 먼저 current->rlim [RLIMIT_NPROC].rlim_cur이 사용자소유로 있는 현재프로쎄스의 개수보다 작거나 같으면 프로쎄스가 뿌리권한이 있지 않은 이상 오유코드를 돌려준다. 사용자가 소유하고있는 현재프로쎄스의 개수는 사용자마다 존재하는 user_struct자료구조에서 얻는다. 이 자료구조는 프로쎄스서술자의 user마당에 있는 지적자를 통해서 찾을수 있다.

⑤ 전체 프로쎄스의 개수가 max_threads변수값보다 작은지 검사한다. 이 변수의 초기값은 체계에 있는 RAM의 크기에 따라 다르다. 일반적인 규칙은 프로쎄스서술자와 핵심부방식탄창이 물리기억기의 1/8이상을 점유하지 못하게 하는것이다. 그러나 체계관리자는 /proc/sys/ kernel/threads-max파일에 값을 써넣어서 이것을 바꿀수 있다.

⑥ 부모프로쎄스가 어떤 핵심부모듈이라도 사용하고있다면 해당 참고회수를 증가시킨다. 매 핵심부모듈은 자신의 참조회수를 가지며 이를 통해 사용중인 모듈을 제거할수 있게 한다.

⑦ 부모프로쎄스에서 복사한 flags마당에 있는 기발중 몇가지를 갱신한다.

ㄱ. 프로쎄스가 초사용자권한을 사용했는지를 나타내는 PF_ SUPERPRIV기발을 지운다.

ㄴ. PF_USEDFPU기발을 지운다.

ㄷ. 자식프로쎄스가 아직까지 execve()체계호출을 실행한 사실이 없음을 의미하는 PF_FORKNOEXEC기발을 설정한다.

⑧ 이제 do_fork()함수는 부모프로쎄스에서 얻을수 있는것의 대부분을 얻었다. 나머지 작업은 자식프로쎄스에 있는 자원을 설정하고 핵심부에 새로운 프로쎄스가 탄생했음을 알려주는데 초점을 맞춘다. 먼저 get_pid()함수를 호출해서 자식에 할당할 새로운 PID를 얻는다.(CLONE_PID기발을 설정하지 않은 경우)

⑨ 프로쎄스의 부모관계를 나타내는 마당처럼 부모프로쎄스에서 상속할수 없는 모든 프로쎄스서술자의 마당을 갱신한다.

⑩ flags변수를 리용해서 다르게 설정하지 않았다면 copy_files(), copy_fs(), copy_sighand(), copy_mm()을 호출해서 새로 자료구조를 생성하고 해당 부모프로쎄스자료구조에서 값을 복사한다. (kernel\fork.c)

copy_thread()를 호출해서 자식프로쎄스의 핵심부방식탄창을 clone() 체계호출을 실행한 당시 CPU등록기에 들어있던 값으로 초기화한다.(이 값은 부모의 핵심부방식탄창에 저장되여있다.)

eax등록기에 해당하는 마당을 0으로 만든다. 자식프로쎄스의 서술자에 있는 thread.esp마당을 자식의 핵심부방식탄창의 기본주소로 초기화하고 thread.eip마당에 ret_from_fork()기호언어함수의 주소를 저장한다. copy_thread()함수는 부모프로쎄스에 대해 unlazy_fpu()를 호출하여 thread.i387마당을 복제한다.

⑪ CLONE_THREAD나 CLONE_PARENT중 하나라도 설정하였다면 부모의 p_opptr와 p_pptr마당을 자식의 해당 마당으로 복제한다. 따라서 현재프로쎄스의 부모가 새로 만들어지는 자식프로쎄스의 부모도 된다. 기발을 설정하지 않은 경우 자식프로쎄스의 P_OPPTR과 P_PTR마당을 current의 프로쎄스서술자주소로 설정한다.

CLONE_PTRACE기발을 설정하지 않았다면 자식프로쎄스서술자의 ptrace마당을 0으로 설정한다. 이 마당에는 한 프로쎄스를 다른 프로쎄스가 추적할 때 사용하는 기발 몇개가 들어있다. 현재프로쎄스를 추적하고있더라도 자식프로쎄스는 추적하지 않을것이다.

반대로 CLONE_PTRACE기발을 설정한 경우 부모프로쎄스를 추적중인지를 검사한다. 이 경우라면 자식프로쎄스 역시 추적해야 한다. 따라서 current->ptrace에서 PT_PTRACED가 설정되여있으면 current->p_pptr를 자식의 해당 마당으로 복사한다.

CLONE_THREAD값을 검사한다. 이 기발을 설정한 경우 자식을 부모의 스레드그룹에 포함하고 부모의 tgid마당값을 해당 마당으로 복사하고 그렇지 않은 경우 tgid마당을 pid마당값으로 설정한다.

SET_LINKS마크로를 사용하여 새로운 프로쎄스서술자를 프로쎄스목록에 삽입한다.

hash_pid()를 호출하여 새로운 프로쎄스서술자를 pidhash하쉬표에 추가한다.

nr_threads와 current->user->processes값을 증가시킨다.

자식프로쎄스를 추적하고있으면 이 프로쎄스에 SIGSTOP신호를 보내서 프로쎄스를 시작하기 전에 오유추적기가 먼지 이것을 살펴볼수 있게 한다.

wake_up_process()를 호출하여 자식프로쎄스서술자의 state마당을 TASK_RUNNING으로 설정하고 자식을 실행대기렬목록에 넣는다.

⑫ CLONE_VFORK기발을 설정한 경우 부모프로쎄스를 대기렬에 넣고 자식프로쎄스가 자신의 기억기주소공간을 해제할 때까지(즉 자식프로쎄스가 완료하거나 새로운 프로그람을 실행할 때까지) 부모프로쎄스를 멈추게 한다.

⑬ 자식프로쎄스의 PID를 반환한다. 사용자방식에 있는 부모프로쎄스는 최종적으로 이 PID값을 받는다.

이제 실행가능한 상태인 완벽한 자식프로쎄스를 만들었다. 그러나 이것을 실제로 실행하고있지는 않다. 이 자식프로쎄스에 언제 CPU를 할당할지는 순서짜기프로그람이 결정할 일이다. 언젠가는 자식프로쎄스로 절환하여 자식프로쎄스서술자에 있는 thread마당값으로 몇개의 CPU등록기를 설정할것이다. 특히 esp를 thread.esp(즉 자식의 핵심부방식탄창의 주소)로, eip를 ret_from_fork()의 주소로 설정할것이다. 이 기호언어함수는 ret_from_sys_call()함수를 호출하여 다른 모든 등록기를 탄창에 저장한 값으로 복구한 후 CPU를 사용자방식으로 돌려보낸다. 그러면 새 프로쎄스는 fork()또는 vfork(), clone()체계호출의 끝에서 실행을 시작한다. 체계호출이 반환하는 값은 eax에 들어있다. 이 값은 자식일 경우 0, 부모일 경우 자식의 PID이다.

자식프로쎄스는 fork함수가 0을 반환하는것을 제외하고는 부모와 똑같은 코드를 실행한다. Unix프로그람작성자에게는 낯익은 방식인데 응용프로그람개발자는 이 사실을 리용해서 프로그람에 조건문을 넣어 PID값에 따라 자식프로쎄스가 부모와 다르게 동작하도록 할수 있다.

2) 핵심부스레드

전통적인 Unix체계는 디스크캐쉬의 내용을 저장하고 사용하지 않는 페지들을 교환하여내보내기(swap out)하고 망접속을 봉사하는 등의 중요한 작업을 주기적으로 실행하는 프로쎄스에 위임한다. 실제로 이러한 작업을 엄격히 바로 수행하는것은 비효률적이다. 대신 이것을 배경작업으로 순서짜기하면 원하는 기능과 사용자프로쎄스 모두를 좀더 잘 반응할것이다. 체계프로쎄스중 몇가지는 핵심부방식에서만 동작하기때문에 최신조작체계는 이러한 기능을 거치장스럽고 불필요한 사용자방식문맥을 사용하지 않는《핵심부스레드(kernel thread)》에 위임한다. Linux에서 핵심부스레드는 다음과 같은 점에서 정규프로쎄스와 다르다.

- 매 핵심부스레드는 핵심부의 특정C함수 하나만 실행하지만 정규프로쎄스는 체계호출을 통해서만 핵심부함수를 실행한다.
- 핵심부스레드는 핵심부방식에서만 동작하지만 정규프로쎄스는 핵심부방식과 사

용자방식을 번갈아가며 동작한다.

- 핵심부스레드는 핵심부스레드에서만 동작하므로 PAGE_OFFSET 보다 큰 선형주소만 사용한다. 반대로 정규프로세스는 4GB선형주소모두를 사용자방식이나 핵심부방식에서 사용한다.

3) 핵심부스레드생성

kernel_thread()함수는 새 핵심부스레드를 만들며 다른 핵심부스레드에서만 이 함수를 실행할수 있다. 이 함수는 대부분 inline기호언어(assembly language)코드로 되여있다.

4) 프로세스 0

모든 프로세스의 조상은 《 프로세스 0 》 또는 《 교환기프로세스(swapper process)》라고 부르며 start_kernel()함수가 Linux를 초기화하는 과정에서 맨 바닥에서 만드는 핵심부스레드이다. 이 조상프로세스는 다음 자료구조를 사용한다.

・init_task_union변수에 들어있는 프로세스서술자와 핵심부방식탄창, init_task와 init_stack마크로는 각각 프로세스서술자와 탄창의 주소를 만든다.

・프로세스서술자내의 여러 마당에서 가리키는 다음표

-init_mm
-init_fs
-init_files
-init_signals

각각 다음 마크로를 통해 이 표를 초기화한다.

-INIT_MM
-INIT_FS
-INIT_FILES
-INIT_SIGNALS

・swapper_pg_dir에 들어있는 기본(master)페지대역등록부

start_kernel()함수는 핵심부가 필요로 하는 모든 자료구조를 초기화하며 새치기를 허용하고 《init프로세스》로 잘 알려진 《프로세스1》이라는 다른 핵심부스레드를 생성한다.

kernel_thread(init, NULL, CLONE_FS|CLONE_FILES|CLONE_SIGNAL);

새로 만들어진 핵심부스레드는 PID 1이며 프로세스마다 할당하는 모든 핵심부자료구조를 프로세스0과 공유한다.

순서짜기프로그람이 init프로세스를 선택하면 이것은 init()함수를 실행한다.

init프로세스를 만든후 프로세스0은 cpu_idle()함수를 실행한다. 이 함수는 근본적으로 새치기를 허용한 상태에서 반복해서 hlt기호언어명령을 실행한다. 순서짜기프로그람은 TASK_RUNNING상태에 있는 다른 프로세스가 없는 경우에만 프로세스 0을 선

택한다.

5) **프로세스 1**

프로세스0이 생성한 핵심부스레드는 init()함수를 실행하고 이것은 핵심부의 초기화를 완료한다. init()는 execve()체계호출을 실행해서 실행프로그람 init를 적재한다. 그 결과 init핵심부스레드는 프로쎄스마다 할당받는 자신만의 핵심부자료구조를 가진 일반 프로쎄스로 바뀐다. init프로쎄스는 조작체계의 바깥계층을 실현하는 모든 프로쎄스를 생성하고 동작을 감시하기때문에 체계를 완료할 때까지 살아남는다.

6) **다른 핵심부스레드**

Linux는 그 외에도 여러 핵심부스레드를 활용한다. 그 가운데서 어떤것은 초기화단계에서 생성하여 체계를 완료할 때까지 실행하고 어떤것은 핵심부가 별도의 실행문맥에서 실행할 경우 더 좋은 성능을 내는 작업을 실행해야 할 때 《필요에 따라》 생성한다.

가장 중요한 핵심부스레드(프로쎄스0과 프로쎄스1을 제외하고)는 다음과 같다.

keventd

qt_context작업렬(task queue)에 있는 작업을 실행한다.

kapmd

고급전원관리(APM, Advanced Power Management)와 관련있는 사건을 처리한다.

kswapd

기억기를 해제한다.

kflushd(bdflush**라고도 한다**)

《불결한(dirty)》완충기를 디스크에 저장해서 기억기를 비운다.

kupdated

오래된 《불결한》완충기를 디스크에 기록해서 파일체계의 자료가 잘못될 위험을 줄인다.

ksoftirqd

소작업(tasklet)을 실행한다. 체계에 있는 매 CPU마다 이 스레드가 하나씩 있다.

11. 프로쎄스끝내기

대부분의 프로쎄스는 자신이 실행하던 코드의 실행이 끝나면 《죽는다.》이런 일이 일어나면 핵심부가 프로쎄스가 소유하던 자원(이런 자원으로는 기억기와 열린 파일 그리고 신호기 등이 있다)을 해제할수 있도록 핵심부에 알려주어야 한다.

프로쎄스를 완료하는 가장 일반적인 방법은 exit()서고함수를 호출하는것이다. exit()는 C서고가 할당한 자원을 해제하고 프로그람작성자가 등록한 매 함수를 실행하며 _exit()체계호출을 실행하여 끝난다. 프로그람작성자는 명시적으로 exit()함수를 호출할수도 있다. 추가로 C콤파일러는 항상 main() 함수의 마지막문장 바로 뒤에 exit()

함수를 호출하는 코드를 삽입한다.

다른 경우로 핵심부가 프로쎄스를 강제로 죽일수도 있다. 이것은 보통 프로쎄스가 자신이 처리할수 없거나 무시할수 없는 신호를 받거나 어떤 프로쎄스의 문맥에서 핵심부 방식으로 들어가 실행하던 도중 복구할수 없는 CPU례외가 발생한 경우에 일어난다.

1) 프로쎄스완료

모든 프로쎄스완료는 do_exit()함수가 처리하며 이 함수는 핵심부자료구조에서 완료하는 프로쎄스에 관한 대부분의 참조를 제거한다. do_exit()함수는 다음과 같은 작업을 한다.

① 프로쎄스서술자의 flag마당에 PF_EXITING기발을 설정하여 프로쎄스를 제거하는중임을 표시한다.

② 필요하면 sem_exit()함수를 호출하여 IPC신호기대기렬에서 프로쎄스서술자를 제거하고 del_timer_sync()함수를 호출하여 동적시계대기렬에서 프로쎄스서술자를 제거한다.

③ __exit_mm()과 __exit_files, __exit_fs(), __exit_sighand()함수를 호출해서 각각 페지화와 파일체계, 열린파일서술자, 신호처리 등과 관련한 프로쎄스의 자료구조를 점검한다. (kernel\exit.c)

④ 이 함수는 이런 자료구조를 다른 프로쎄스와 더는 공유하지 않으면 이것을 제거하는 일도 한다.

⑤ 프로쎄스가 사용하는 모듈의 자원참조회수를 감소시킨다.

⑥ 프로쎄스서술자의 exit_code마당을 프로쎄스완료코드로 설정한다. 이 값은 _exit()체계호출에 넘어온 값이거나(정상적인 완료) 핵심부가 넘겨준 오유코드이다.(비정상적인 완료)

⑦ exit_notify()함수를 호출해서 부모프로쎄스와 자식프로쎄스 량쪽의 친족관계를 갱신한다. 완료하는 프로쎄스가 만든 자식프로쎄스는 스레드그룹이 있다면 같은 그룹에 있는 다른 프로쎄스의 자식이 되고 없다면 《init》프로쎄스의 자식이 된다.

나아가서 exit_notify()함수는 프로쎄스서술자의 state마당을 TASK_ZOMBIE로 설정한다. 후에 좀비프로쎄스에서 어떤 일이 일어나는가를 보자.

⑧ schedule()함수를 호출해서 실행할 새로운 프로쎄스를 선택한다. 순서짜기프로그람은 TASK_ZOMBIE상태인 프로쎄스를 무시하기때문에 이 프로쎄스는 schedule()함수에서 switch_to마크로를 호출한 직후 실행을 중단한다.

2) 프로쎄스제거

Unix조작체계는 프로쎄스가 핵심부에 부모프로쎄스의 PID나 자식프로쎄스의 실행상태를 질의하여 이것을 알아낼수 있다. 례를 들어 프로쎄스는 특정작업을 수행하는 자식프로쎄스를 만든 다음 이 프로쎄스가 완료했는지를 검사하기 위해 wait()계렬 체계호출을 실행할수 있다. 자식프로쎄스가 완료했으면 부모프로쎄스는 완료코드를 통해서 작

업을 성공적으로 수행했는지 여부를 알수 있다.

이런 설계상의 선택때문에 Unix핵심부는 프로쎄스가 끝난 직후에 이 프로쎄스서술자에 들어있는 자료를 함부로 버릴수 없다. 오로지 부모프로쎄스가 완료한 자식프로쎄스를 지정하여 wait()계렬체계호출을 실행한 후에만 이 자료를 버릴수 있다. 이것이 TASK_ZOMBIE상태가 있는 리유이다. 프로쎄스는 기술적으로는 죽었지만 부모프로쎄스가 이 사실을 알 때까지 프로쎄스의 서술자를 저장하고있어야 한다.

부모프로쎄스가 자식프로쎄스보다 먼저 완료하면 어떻게 되는가? 이런 경우 체계가 좀비프로쎄스로 가득 차서 사용할수 있는 task입구점을 모두 소모해버릴수도 있을것이다. 그러나 앞서 언급한대로 이 문제는 고아가 된 모든 프로쎄스를 《init》프로쎄스의 자식으로 만들어서 해결한다. 이런식으로 《init》프로쎄스는 wait()계렬의 체계호출을 실행해서 자신의 합법적인 자식 프로쎄스중 하나가 완료했는지 검사하여 좀비를 없앤다.

release_task()함수는 다음 과정을 통해 좀비프로쎄스의 프로쎄스서술자를 해제한다.

① 완료한 프로쎄스의 소유자인 사용자가 지금까지 만든 프로쎄스의 수를 하나 줄인다. 이 값은 앞에서 언급한바 있는 user_struct구조체에 들어 있다.

② free_uid()함수를 호출하여 user_struct구조체의 자원참조회수를 1 줄인다.

③ unhash_process()를 호출한다. (kernel\exit.c)

④ 이 함수는 다음과 같은 작업을 수행한다.

ㄱ. nr_threads변수값을 1 줄인다.

ㄴ. unhash_pid()함수를 호출해서 pidhash하쉬표에서 프로쎄스서술자를 제거한다.

ㄷ. REMOVE_LINKS마크로를 사용하여 프로쎄스목록에서 프로쎄스서술자를 떼여낸다.

ㄹ. 스레드그룹에 속해있다면 프로쎄스를 이 그룹에서 제거한다.

⑤ free_task_struct()함수를 호출해서 프로쎄스서술자와 핵심부방식탄창으로 사용한 8kB기억기령역을 해제한다.

제 2 절. 프로쎄스주소공간

핵심부함수는 동적기억기가 필요하면 간단하게 다음 함수중 어느 하나를 호출한다. 형제체계알고리듬에서 페지를 할당받을 때에는 __get_free_pages()나 alloc_pages(), 특별한 객체나 일반적인 객체용으로 종속할당자를 사용할 때에는 kmem_cache_alloc()나 kmalloc(), 불련속적인 기억기령역을 얻을 때에는 vmalloc() 를 호출한다. 이 함수는 기억기요청을 성공적으로 처리하면 할당한 동적기억기령역의 시작위치를 나타내는 페지서술자의 주소나 선형주소를 반환한다.

이렇게 단순한 방법이 잘 동작하는 리유는 다음 두가지이다.

・핵심부는 조작체계에서 우선순위가 가장 높은 구성요소이다. 핵심부함수가 동적기억기를 요청하려면 반드시 정당한 리유가 있어야 하고 이 요청을 더는 뒤로 미룰수 없을 때여야 한다.

・핵심부는 자신을 믿는다. 모든 핵심부함수는 오유가 없다고 가정하기때문에 프로그람오유로부터 보호할 필요가 없다.

그러나 사용자방식의 프로쎄스에 기억기를 할당하는 경우에는 상황이 완전히 달라진다.

・프로쎄스가 요청하는 동적기억기는 급하지 않다고 간주한다. 례를 들어 프로쎄스의 실행파일을 적재하는 경우 가까운 시일내에 프로쎄스코드가 있는 모든 페지주소를 참조하지는 않을것이다. 마찬가지로 프로쎄스가 추가로 동적기억기를 얻으려고 malloc()를 호출하는것은 프로쎄스가 추가로 얻은 기억기전체를 바로 사용함을 의미하지는 않는다. 따라서 일반적으로 핵심부는 사용자방식프로쎄스에 동적기억기를 할당하는 일을 미루려고 한다.

・사용자프로그람은 믿을수 없기때문에 핵심부는 사용자방식에서 프로쎄스가 일으킬수 있는 모든 주소오유를 잡아낼수 있도록 준비해야 한다.

이 절에서 보지만 핵심부는 새로운 종류의 자원을 리용하여 프로쎄스에 동적기억기를 할당하는 일을 미룰수 있다. 사용자방식프로쎄스가 동적기억기를 요청하면 핵심부는 페지틀을 추가로 할당해주지 않고 선형주소의 새로운 범위를 사용할수 있는 권한을 주는데 이 범위는 프로쎄스주소공간의 일부가 된다. 이러한 구간을 가리켜 《기억기구역(memory region)》이라고 한다.

1. 프로쎄스의 주소공간

프로쎄스의 《주소공간(address space)》은 프로쎄스가 사용할수 있는 모든 선형주소로 구성된다. 매 프로쎄스는 서로 다른 선형주소공간집합을 바라보며 한 프로쎄스에서 사용하는 주소는 다른 프로쎄스에서 사용하는 주소와 아무 관련이 없다. 후에 보지만 선형주소구간을 추가하거나 삭제하여 프로쎄스의 주소공간을 동적으로 바꿀수 있다.

핵심부는 선형주소구간의 시작선형주소와 길이, 몇가지 접근권한을 서술하는 《기억기구역(memory region)》이라는 자원으로 선형주소구간을 표현한다. 효률성을 위해 기억기구역의 시작주소와 길이는 모두 4096의 배수여야 한다. 따라서 매 기억기구역으로 구별하는 자료로 할당받은 페지틀을 모두 채울수 있다. 다음은 프로쎄스에 새로운 기억기구역을 할당하는 전형적인 상태이다.

・사용자가 콘솔에 명령을 내리면 쉘프로쎄스는 명령을 수행할 새로운 프로쎄스를 만든다. 그 결과 새로 만든 프로쎄스에 새로운 주소공간 즉 일련의 기억기구역을 할당한다.

・실행중인 프로쎄스는 완전히 다른 프로그람을 적재하려 할수도 있다. 이 경우

프로쎄스ID는 바뀌지 않지만 프로그람을 적재하기 전에 사용하던 기억기구역을 해제하고 프로쎄스에 새로운 기억기구역집합을 할당한다.

· 실행중인 프로쎄스는 파일(또는 파일의 일부)에 《 기억기배치(memory mapping)》를 할수도 있다. 이 경우 핵심부는 파일을 배치하기 위해 프로쎄스에 새로운 기억기구역을 할당한다.

· 프로쎄스가 사용자방식탄창에 자료를 계속 추가하여 후에 탄창으로 배치한 기억기구역의 주소를 모두 사용해버릴수도 있다. 이 경우 핵심부는 이 기억기구역을 확장하겠다고 결정할수 있다.

· 프로쎄스는 같이 일하는 다른 프로쎄스와 자료를 공유하려고 IPC공유 기억기구역을 만들수도 있다. 이 경우 핵심부는 이러한 구조를 실현할수 있도록 프로쎄스에 새로운 기억기구역을 할당한다.

· 프로쎄스는 malloc()와 같은 함수를 리용하여 자신의 동적령역을 확장할수 있다. 핵심부는 이에 따라 동적기억구역으로 할당한 기억기구역의 크기를 확장하겠다고 결정할수 있다.

표 3-1은 앞에서 언급한 작업과 관련한 몇가지 체계호출을 보여준다. 이 장 끝에서 설명하는 brk()를 제외한 나머지 체계호출은 후에 설명한다.

표 3-1. 기억기구역생성, 삭제와 관련한 체계호출

체계호출	설 명
brk(), sbrk()	프로쎄스의 동적기억구역크기를 바꾼다.
execve()	새로운 실행파일을 적재하는데 그 결과 프로쎄스의 주소공간이 바뀐다.
_exit()	현재프로쎄스를 완료하고 프로쎄스의 주소공간을 없앤다.
fork()	새로운 프로쎄스를 생성하여 새 주소공간을 만든다.
mmap()	파일에 대한 기억기배치를 만들어 프로쎄스주소공간을 늘인다.
Munmap()	파일에 대한 기억기배치를 없애서 프로쎄스주소공간을 줄인다.
shmat()	공유기억기구역을 만든다.
shmdt()	공유기억기구역을 없앤다.

교육성 교육정보쎈터

페지오유례외조종기에서 보지만 핵심부는 프로쎄스가 현재 소유하는 기억기구역을 반드시 파악하고있어야 한다. 그래야 페지오유례외조종기가 다음과 같은 페지오유가 발생한 잘못된 선형주소의 두가지 류형을 효률적으로 구별할수 있다.

· 프로그람오유때문에 발생한 경우

· 페지틀을 할당하지 않아서 발생한 경우. 선형주소가 프로쎄스주소공간에 들어있어도 이 주소에 해당하는 페지틀을 아직 할당하지 않았다.

후자의 경우에 해당하는 주소는 프로쎄스립장에서 보면 잘못된 주소가 아니다. 핵심부는 페지틀을 제공한 후 프로쎄스를 계속 실행하여 이 페지오유를 처리한다.

2. 기억기서술자

프로쎄스주소공간과 관련한 모든 정보는 《기억기서술자(memory descriptor)》라는 자료구조에 들어있다. 이 구조체의 형태는 mm_struct고 프로쎄스서술자의 mm마당이 이것을 가리킨다. 표 3-2는 기억기서술자에 있는 마당의 항목이다.

표 3-2. 기억기서술자의 마당

형	마당	설명
struct vm_area_struct *	mmap	기억기구역객체목록의 머리를 가리키는 지적자
rb_root_t	mm_rb	기억기구역객체의 빨간색-검은색나무 (red-block tree)의 뿌리마디에 대한 지적자
Struct vm_area_struct *	mmap_cache	마지막으로 참조한 기억기구역객체를 가리키는 지적자
pgt_t *	pgd	페지대역등록부를 가리키는 지적자
atomic_t	mm_users	2차사용회수
atomic_t	mm_count	기억기구역의 수
struct vw_semaphore	mmap_sem	기억기구역의 읽기/쓰기 신호기
spinlock_t	page_table_lock	기억기구역과 페지표의 스핀잠그기
Struct list_head	mmlist	기억기서술자목록에서 린접하는 항목을 가리키는 지적자
unsigned long	start_code	실행코드의 시작주소

unsigned long	end_code	실행코드의 끝주소
unsigned long	start_data	초기화된 자료의 시작주소
unsigned long	end_data	초기화된 자료의 끝주소
unsigned long	start_brk	동적기억구역의 시작주소
unsigned long	brk	동적기억구역의 현재 끝주소
unsigned long	start_stack	사용자방식탄창의 시작주소
unsigned long	arg_start	명령행인자의 시작주소
unsigned long	arg_end	명령행인자의 끝주소
unsigned long	env_start	환경변수의 시작주소
unsigned long	env_end	환경변수의 끝주소
unsigned long	rss	프로쎄스에 할당한 페지들의 수
unsigned long	total_vm	프로쎄스주소공간의 크기(페지수)
unsigned long	locked_vm	교체해 내보내지(swap out) 못하도록 《잠그기를 건》페지의 수
unsigned long	def_flags	기억기구역의 기본접근 기발
unsigned long	cpu_vm_mask	지연TLB를 위한 비트마스크
unsigned long	swap_address	교체를 하려고 마지막으로 검사한 선형 주소
unsigned long	dumpable	프로쎄스가 기억기의 핵심쏟기(core dump) 파일을 만들수 있는지를 나타내는 기발
unsigned long	context	기본방식에 고유한 정보(례를 들어 80x86 기반에서 LDT의 주소)를 저장하는 표를 가리키는 지적자

모든 기억기서술자를 2중련결목록으로 저장한다. 매 서술자는 mmlist마당에 린접하

는 항목의 주소를 저장한다. 목록의 첫번째 항목은 프로쎄스 0이 초기화 단계에서 사용하는 기억기서술자인 init_mm의 mmlist마당이다. 이 목록을 다중처리기체계에서 동시에 접근하는것을 막으려고 mmlist_lock스핀잠그기를 사용한다. 체계에 있는 기억기서술자의 개수는 mmlist_nr변수에 저장한다.

mm_users마당은 mm_struct자료구조를 공유하는 가벼운 프로쎄스의 개수이다. mm_count마당은 기억기서술자의 1차사용회수이고 mm_users에 있는 모든 《사용자》를 mm_count에서는 한 단위로 센다. 핵심부는 mm_count마당을 감소시킬 때마다 이 값이 0이 되는지 검사하여 0이 되면 더는 사용하지 않는것이므로 기억기서술자를 해제한다.

mm_users와 mm_count용도가 어떻게 다른지 례를 들어 설명한다. 두 가벼운 프로쎄스가 기억기서술자하나를 공유한다고 하자. 일반적으로 mm_user마당은 값이 2이지만 mm_count마당은 1이다.(이것을 소유하는 두 프로쎄스를 하나로 센다.)

기억기서술자를 림시로 핵심부스레드에 빌려준다면 핵심부는 mm_count마당을 증가시킨다. 이렇게 해서 두 가벼운 프로쎄스가 완료하여 mm_users마당이 0이 되더라도 핵심부스레드가 이것을 사용하는 동안에는 mm_count마당이 0보다 값이 크므로 해당 기억기 서술자를 해제하지 않는다. 핵심부는 시간이 오래 걸리는 작업을 하는 도중에 기억기서술자를 해제하지 않고싶다면 mm_count마당대신 mm_usere마당을 증가시킬수 있다.(바로 swap_out()함수가 하는 일이다.) mm_users마당을 증가시키면 해당 기억기서술자를 소유하는 모든 가벼운 프로쎄스가 완료하더라도 mm_count가 0이 되지 않음을 보장하므로 최종결과는 똑같다.

새 기억기서술자를 얻을 때에는 mm_alloc()함수를 호출한다.

이 기억기서술자를 종속할당자캐쉬에 저장하기때문에 mm_alloc()는 kmem_cache_alloc()를 호출한 후 새로운 기억기서술자를 초기화하고 mm_count와 mm_users마당을 1로 설정한다.

반대로 mmput()함수는 기억기서술자의 mm_users마당을 감소시킨다. 이 마당이 0이 되면 이 함수는 지역서술자표와 기억기구역서술자, 기억기서술자가 참조하는 페지표를 해제한 후 mmdrop()를 호출한다.

mmdrop()함수는 mm_count를 감소시키고 이 값이 1이 되면 mm_struct자료구조를 해제한다.

mmap, mm_rb, mmlist, mmap_cache마당은 다음에 설명한다.

핵심부스레드의 기억기서술자

핵심부스레드는 핵심부방식에서만 동작하기때문에 TASK_SIZE(PAGE_ OFFSET와 같다. 보통 0xc0000000)아래에 있는 선형주소에 접근하지 않는다. 정규프로쎄스와는 반대로 핵심부스레드는 기억기구역을 사용하지 않으므로 기억기서술자에 있는 대부분의 마당은 핵심부스레드에 의미가 없다.

TASK_SIZE우의 선형주소를 참조하는 페지표입구점은 항상 똑같아야 하므로 핵심

부스레드가 어떤 페지표집합을 사용하는지는 문제가 되지 않는다. Linux 핵심부에서 핵심부스레드는 쓸데없이 TLB와 캐쉬를 비우지 않기 위해 정규프로쎄스의 페지표를 그대로 사용한다. 이것을 위해 모든 프로쎄스서술자에는 mm과 active_mm이라는 두 종류의 기억기서술자가 있다.

프로쎄스서술자의 mm마당은 프로쎄스가 소유하는 기억기서술자, active_mm 마당은 프로쎄스가 실행중에 사용하는 기억기서술자를 가리킨다. 정규 프로쎄스의 경우 두 마당이 같은 곳을 가리키지만 핵심부스레드는 기억기서술자를 포함하지 않으므로 mm마당은 항상 NULL이다. 핵심부스레드를 선택하여 실행할 때 active_mm마당을 바로 전에 실행하던 프로쎄스의 active_mm마당의 값으로 초기화한다.

그런데 여기서 약간의 문제가 있다. 핵심부스레드에서 실행중인 프로쎄스가 《높은》 선형주소(TASK_SIZE 우)에 해당하는 페지표입구점을 수정하면 체계에 있는 모든 프로쎄스의 페지표집합에 있는 해당 입구점 또한 갱신해야 한다. 사실 핵심부방식에 있는 어떤 프로쎄스든지 페지표입구점을 설정하면 이 기억기배치는 핵심부방식에 있는 다른 모든 프로쎄스에서도 유효해야 한다. 모든 프로쎄스에 있는 페지표집합을 다루는데는 시간이 많이 걸리므로 Linux는 이것을 미루는 접근방법을 채택한다.

높은 선형주소의 배치를 바꿀 때마다(대체로 vmalloc()나 vfree()를 사용하여) 핵심부는 기본핵심부페지대역등록부(master kernel page global directory)인 swapper_pg_dir를 뿌리로 하는 표준페지표를 갱신한다. (mm\vmalloc.c)

init_mm변수에 들어 있는 《주기억기서술자(master memory descriptor)》의 pgd마당은 이 페지대역등록부를 가리킨다.

3. 기억기구역

Linux는 vm_area_struct형태의 객체를 리용하여 기억기구역을 구현한다.

vm_area_struct구조체 내용을 보면 다음과 같다.

```
struct vm_area_struct {
    struct mm_struct * vm_mm;      /* 주소공간. */
    unsigned long vm_start;        /* 시작주소. */
    unsigned long vm_end;
    /*과제당VM 령역의 련결된목록 (주소로 저장)*/
    struct vm_area_struct *vm_next;

    pgprot_t vm_page_prot;         /* VMA의 접속권한. */
    unsigned long vm_flags;
    struct rb_node vm_rb;
```

```
    union {
            struct {
                    struct list_head list;
                    void *parent;       /* prio_tree_node부모 */
                    struct vm_area_struct *head;
            } vm_set;

            struct prio_tree_node prio_tree_node;
    } shared;

    struct list_head anon_vma_node;       /* anon_vma->lock묶음 */
    struct anon_vma *anon_vma;      /* page_table_lock */
    /*이 구조체를 넘겨주기 위한 포인터. */
    struct vm_operations_struct * vm_ops;
    /* 여벌복사정보: */
    unsigned long vm_pgoff;         /* PAGE_SIZE내에서 편위*/
    struct file * vm_file;          /* 매핑 할 파일 (NULL이 될수 있다.) */
    void * vm_private_data;         /* vm_pte (공유기억) */
#ifdef CONFIG_NUMA
    struct mempolicy *vm_policy;    /* VMA을 위한 NUMA방책 */
#endif
};
```

매 기억기구역서술자는 선형주소구간을 구별한다. vm_struct마당은 구간의 첫번째 선형주소를, vm_end마당은 구간이 끝난 직후의 첫번째 선형주소를 저장한다. 그러므로 vm_end-vm_start는 기억기구역의 길이가 된다. vm_mm마당은 구역을 소유하는 프로쎄스의 mm_struct기억기서술자를 가리킨다. vm_area_struct의 나머지 마당은 뒤에서 설명한다.

표 3-3은 이 구조체에 있는 마당의 목록이다.

표 3-3. 기억기구역 객체의 마당

형	마 당	설 명
struct mm_start *	Vm_mm	해당 구역을 소유하는 기억기서술자를 가리키는 지적자
unsigned long	Vm_start	구역내에 있는 첫번째 선형주소

unsigned long	Vm_end	구역이 끝난후의 첫번째 선형주소
struct vm_area_strcut *	Vm_next	프로쎄스의 구역목록에서 다음 구역
pgprot_t	Vm_page_prot	해당 구역에 있는 페지들의 접근권한
unsigned long	Vm_flags	구역의 기발
Rb_node_t	Vm_rb	빨간색-검은색나무용 자료
struct vm_area_struct *	vm_next_share	파일기억기배치목록에 있는 다음 항목을 가리키는 지적자
struct vm_area_struct**	vm_pprev_share	파일기억기배치목록에 있는 이전 항목을 가리키는 지적자
struct vm_operation_struct*	vm_ops	기억기구역의 메쏘드에 대한 지적자
unsigned long	vm_pgoff	파일을 배치하고있으면 배치하는 파일에서의 편위
struct file *	vm_file	파일을 배치하고있으면 배치하는 파일객체를 가리키는 지적자
unsigned long	vm_raend	배치하는 파일의 현재 미리읽기 윈도우의 끝
void *	vm_private_data	해당 기억기구역만을 위한 자료에 대한 지적자

한 프로쎄스가 소유하는 기억기구역들이 서로 겹치는 일은 없으며 핵심부는 기존의 구역 바로 뒤에 새로운 구역을 할당할 때 이 구역을 합치려고 한다. 린접한 두 구역은 접근권한이 일치하면 합칠수 있다.

그림 3-7에서 볼수 있는것처럼 새로운 선형주소범위를 프로쎄스의 주소공간에 추가할 때 핵심부는 기존의 기억기구역을 확장할수 있는가를 검사한다.(a경우) 그렇지 않으면 새로운 기억기구역을 생성한다.(b경우) 마찬가지로 프로쎄스의 주소공간에서 선형주소범위를 제거할 때 핵심부는 이에 영향을 받는 기억기구역의 크기를 조정한다.(c경우) 어떤 경우에는 크기를 조정할 때 한 기억기구역이 두개의 작은 구역으로 나누이기도 한다.(d경우)

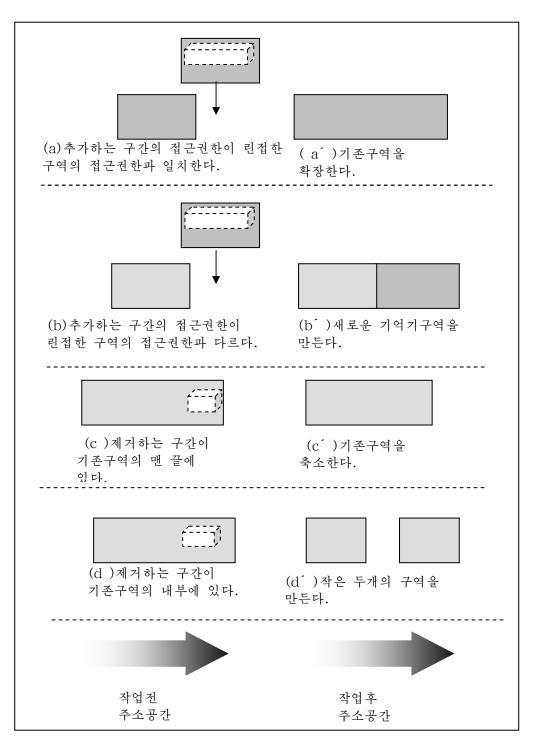



그림 3-7. 선형주소구간의 추가와 제거

vm_ops마당은 기억기구역의 메쏘드를 저장하는 vm_operations자료구조체를 가리킨다. 현재 3가지 메쏘드만을 정의한다.

open

기억기구역을 프로쎄스소유의 기억기구역집합에 추가할 때 호출한다.

close

기억기구역을 프로쎄스소유의 기억기구역집합에서 제거할 때 호출한다.

nopage

프로쎄스가 이 기억기구역에 속하는 선형주소이지만 RAM에 존재하지 않는 페지에 접근하려고 할 때 페지오유조종기가 호출된다.(뒤에 나오는 《페지오유조종기》참고)

1) 기억기구역자료구조

프로쎄스가 소유하는 모든 구역을 단순목록으로 서로 련결한다. 구역은 기억기주소순서에 따라 낮은 주소부터 차례로 목록에 나타난다. 그러나 두 구역사이에 사용하지 않는 기억기주소령역이 들어갈수 있다.

매 vm_area_struct의 vm_next마당은 목록에 있는 다음 항목을 가리킨다. 핵심부는 프로쎄스의 기억기서술자에 있는 mmap마당을 통하여 기억기구역을 찾는다. 이 마당은 목록의 첫번째 기억기구역서술자를 가리킨다.

기억기서술자의 map_count마당은 프로쎄스가 소유한 구역의 개수를 나타낸다. 프로쎄스는 MAX_MAP_COUNT까지 서로 다른 기억기구역을 소유할수 있다.(이 값은 보통 65536이다.)

그림 3-8은 프로쎄스의 주소공간과 기억기서술자, 기억기구역목록사이의 관계를 보여준다.

핵심부가 자주 하는 일들가운데서 하나는 특정선형주소를 포함하는 기억기구역을 찾는것이다. 목록은 정렬되여있으므로 기억기구역의 끝이 지정한 선형주소다음에 있는 구역을 찾는 즉시 검색을 끝마친다.

그러나 목록을 사용하는것은 프로쎄스가 매우 적은(수십개 이하) 기억기구역을 소유하고있을 때에만 편리하다. 목록에서 항목을 찾고 추가 또는 삭제하는데 걸리는 시간은 목록의 길이에 비례한다.

교육성 교육정보쎈터

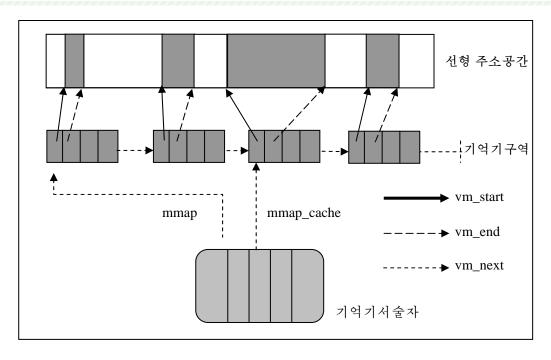



그림 3-8. 프로쎄스의 주소공간과 관련한 여러가지 서술자

대부분의 Linux프로쎄스는 매우 적은 기억기구역만 사용하지만 객체지향자료기지처럼 수백~수천구역을 소유하는 큰 응용프로그람도 있다. 이런 경우 기억기구역목록을 관리하는것은 매우 비효률적이고 기억기와 관련한 체계호출의 성능은 완전히 떨어진다.

그래서 Linux 2.6은 《빨간색-검은색나무(red-black tree)》라는 자료구조에 기억기서술자를 저장한다. 빨간색-검은색나무에서 각 요소(즉 마디(node))는 보통 왼쪽자식(left child)과 오른쪽자식(right child)이라는 두 자식을 가진다. 나무에 있는 요소는 정렬되여있다. 매 마디 N에 대해 N의 왼쪽자식을 근본으로 하는 보조나무의 모든 요소는 N의 오른쪽자식을 근본으로 하는 보조나무(subtree)의 모든 요소와 N의 이전값을 포함한다.(그림 3-9 참고) 마디자체에 적은 수자가 마디의 열쇠값이다.

빨간색-검은색나무는 다음과 같은 4가지 규칙을 준수해야 한다.

1. 모든 마디는 빨간색이나 검은색중의 하나이다.
2. 나무의 뿌리(root)는 검은색이다.
3. 빨간 마디의 자식은 검은색이다.
4. 어떤 마디에서 후대잎(leaf)에 이르는 모든 길에는 같은수의 검은색마디가 있어야 한다. 검은 마디의 수를 셀 때 NULL지적자도 검은 마디로 계산한다.

이 4가지 규칙에 따라 n개의 마디를 포함하는 모든 빨간색-검은색나무는 높이가 최대 2log(n+1)이다.

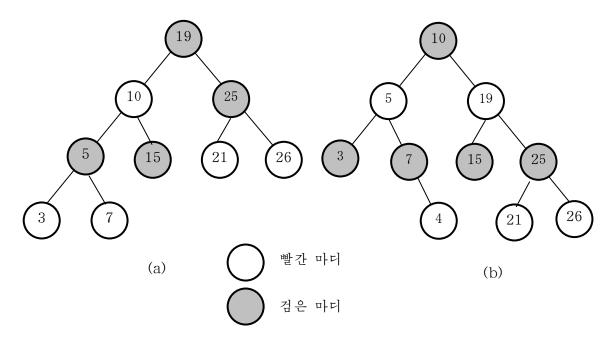



그림 3-9. 빨간색-검은색나무의 례

따라서 빨간색-검은색나무에서 요소를 찾는 작업에 필요한 수행시간은 나무크기에 로그를 취한 값에 그대로 비례하기때문에 매우 효률적이다. 다시 말해서 기억기구역의 수가 2배가 되더라도 찾는 작업을 한번만 더 반복하면 된다.

빨간색-검은색나무는 빠르게 검색하면서 요소를 추가할 곳이나 삭제할 곳을 찾을수 있으므로 요소의 추가와 삭제 또한 효률적이다. 새로운 마디는 빨간 마디에만 삽입해야 한다. 추가나 삭제작업을 하다가 규칙을 파괴하게 되면 나무에 있는 마디 몇개를 옮기거나 색갈을 다시 정해야 한다.

례를 들어 그림 3-9(a)에 나오는 빨간색-검은색나무에 값이 4인 요소를 추가한다고 하자. 이 요소가 들어갈 적당한 위치는 열쇠값이 3인 마디의 오른쪽 자식이지만 여기에 요소를 추가하면 열쇠값이 3인 빨간마디는 빨간마디를 자식으로 가지게 되여 규칙3을 파괴하게 된다. 규칙에 부합하기 위해 값이 3과 4, 7인 마디의 색갈을 바꾼다. 그런데 이런 연산은 규칙 4를 파괴하게 되여 알고리듬에 의해 열쇠값이 19인 마디를 뿌리로 하는 보조나무에 대해 《회전》을 실행하여 그림 3-9에서 (b)와 같은 빨간색-검은색나무를 만든다. 이 작업은 복잡하게 보이지만 빨간색-검은색나무에 요소를 추가하고 삭제하는데 필요한 연산회수는 얼마되지 않는다.(나무크기에 로그를 취한 값에 정비례한다.)

Linux는 프로쎄스의 기억기구역을 저장하기 위해 련결목록과 빨간색-검은색나무를 같이 사용한다. 두 자료구조는 모두 같은 기억기구역서술자를 가리킨다. 기억기구역서술자를 추가하거나 삭제할 때 핵심부는 빨간색-검은색나무를 통해 이전요소와 다음요소를

빠르게 검색하고 이것을 사용하여 련결목록을 검색하지 않고도 련결목록을 빠르게 갱신할수 있다.

기억기서술자의 mmap마당은 련결목록의 머리를 가리킨다. 모든 기억기구역객체는 목록에서 다음요소에 대한 지적자를 vm_next마당에 저장한다. 기억기서술자의 mm_rb 마당은 빨간색-검은색나무의 머리를 가리킨다. 모든 기억기구역객체는 rb_node_t형의 vm_rb마당에 마디의 색갈과 함께 부모와 왼쪽 자식, 오른쪽 자식에 대한 지적자를 저장한다.

일반적으로 특정한 주소를 포함하는 구역을 찾을 때 빨간색-검은색나무를 사용하며 전체 구역을 검색할 때 련결목록이 효과적이다.

2) 기억기구역접근권한

기억기구역의 페지와 관련된 기발을 보기로 하자. 이 기발은 vm_area_struct 서술자의 vm_flags마당에 들어간다.(표 3-4를 참고) 이 기발의 일부는 해당 구역에 들어있는 내용과 프로쎄스가 매 페지에 접근할 때 필요한 권한같은 기억기구역에 있는 모든 페지에 대한 핵심부정보를 제공한다. 나머지 기발은 구역이 어떻게 확장하는가 등의 기억기구역자체를 서술한다.

표 3-4. 기억기구역 기발

기발 이름	설 명
VM_READ	읽을수 있는 페지
VM_WRITE	쓸수 있는 페지
VM_EXEC	실행할수 있는 페지
VM_SHARED	여러 프로쎄스가 공유할수 있는 페지
VM_MAYREADV M_READ	기발을 설정할수 있다.
VM_MAYWRITEVM_WRITE	기발을 설정할수 있다.
VM_MAYEXECVM_EXEC	기발을 설정할수 있다.
VM_GROWSDOWN	이 구역은 낮은 주소쪽으로 확장할수 있다.
VM_GROWSUP	이 구역은 높은 주소쪽으로 확장할수 있다.
VM_SHM	IPC공유기억기로 사용할수 있는 페지
VM_DENYWRITE	이 구역은 쓰기용으로 열수 없는 파일을 배치한다.
VM_EXECUTABLE	이 구역은 실행파일을 배치한다.

VM_LOCKED	이 구역에 있는 페지를 잠그어놓아서 교체해 내보낸다.
VM_IO	이 구역은 장치의 입출력(I/O)주소공간을 배치한다.
VM_SEQ_READ	응용프로그람은 페지를 순차적으로 접근한다.
VM_RAND_READ	응용프로그람은 페지를 완전히 임의의 순서로 접근한다.
VM_DONTCOPY	fork()로 새로운 프로세스를 만들 때 이 구역을 복사하지 않는다.
VM_DONTEXPAND	mremap()체계호출을 사용하여 구역을 확장하는것을 금지한다.
VM_RESERVED	이 구역을 교체해 내보내지 않는다.

기억기구역서술자에 들어가는 페지접근권한을 마음대로 결합할수 있다. 례를 들어 기억기구역에 있는 페지를 실행할수 있지만 읽지 못하게 할수도 있다. 이런 보호대책을 효률적으로 실현하려면 기억기구역의 페지와 관련한 읽기, 쓰기, 실행접근권한을 해당 페지표입구점으로 복사하여 페지화기구회로가 직접 권한검사를 하게 해야 한다. 다시 말해서 페지접근권한은 어떤 종류의 접근을 할 때 페지오유례외가 발생해야 하는가를 지정하는 곳이다. Linux는 페지오유가 발생한 리유를 알아내는 작업을 페지오유조종기에 위임한다. 페지오유조종기는 페지를 다루는 여러 전략을 실현한다.

vm_area_struct서술자의 vm_page_prot마당은 페지표기발의 초기값을 저장한다.(이미 본것처럼 기억기구역에 있는 모든 페지는 기발이 같아야 한다.) 핵심부는 페지를 추가할 때 vm_page_prot마당값에 따라 해당 페지표입구점의 기발을 설정한다.

그러나 다음과 같은 리유때문에 기억기구역의 접근권한을 하드웨어적으로 사용하는 페지보호비트(page protection bit)로 변환하는것은 간단하지 않다.

- 기억기구역의 vm_flags마당에서 지정하는 페지접근권한에 따라 허가된 접근을 하더라도 페지를 접근할 때 페지오유례외가 발생해야 하는 경우가 있다. 례를 들어 후에 《쓰기복사》에서 설명한것처럼 서로 다른 두 프로세스에 소속되였지만 내용이 똑같은 쓰기가능한 페지(VM_SHARE 기발을 설정하지 않은)를 페지를 하나에 저장하는 경우이다. 이런 경우 두 프로세스중 하나라도 페지의 내용을 수정하려고 하면 례외가 발생해야 한다.
- 80x86처리기에서 페지표는 Read/Write(읽기/쓰기)기발과 User/Supervisor(사용자/관리자)기발이라는 두 보호비트만 포함한다. 게다가 항

상 사용자방식프로세스가 기억기구역에 들어있는 모든 페지를 접근할수 있어야 하므로 User/Supervisor기발을 항상 설정해야 한다.

Linux는 80x86극소형처리기의 하드웨어적인 제한을 극복하려고 다음과 같은 규칙을 채택하였다.

- 읽기접근권한은 항상 실행접근권한을 포함한다.
- 쓰기접근권한은 항상 읽기접근권한을 포함한다.

게다가 《쓰기복사》기법을 통해 페지들의 할당을 제대로 연기하려고 해당 페지들을 여러 프로쎄스가 공유하면 안될 때마다 페지들을 쓰기금지로 만든다. 그러므로 읽기, 쓰기, 실행, 공유 이렇게 네 접근권한의 가능한 16개의 조합은 실제로 다음 세개로 줄어든다.

- 페지가 쓰기와 공유접근권한을 동시에 소유하면 Read/Write 비트를 1로 설정한다.
- 페지가 읽기나 실행접근권한을 소유하지만 쓰기나 공유권한이 없으면 Read/Write 비트를 0으로 지운다.
- 페지가 어떤 접근권한도 없으면 Present비트를 지워서 페지에 접근할 때마다 무조건 페지오유례외를 발생시킨다. 그러나 Linux는 실제로 페지가 존재하지 않는 경우와 구별하려고 page size(페지크기)비트를 1로 설정한다.

protection_map배렬은 각 접근권한조합에 대응하는 규모가 축소된 보호비트를 저장한다.

3) 기억기구역다루기

기억기를 다루는 자료구조와 상태정보를 기본적으로 리해하였으므로 여기서는 기억기구역서술자를 가지고 동작하는 저수준함수를 보기로 하자. 이 함수는 do_mmap()와 do_munmap()의 실현을 간단하게 만들어주는 보조함수로 생각하면 된다. 두 함수는 각각 프로쎄스의 주소공간을 확장하거나 축소하는 함수로서 뒤에 나오는 《선형주소구간할당》과 《선형주소구간 해제》에서 설명한다. 여기서 다루는 함수보다 높은 수준에서 작업하는 함수는 기억기구역을 변수로 받지 않고 선형주소구간의 시작주소와 길이 접근권한을 변수로 받는다.

(1) 지정한 주소에서 가장 가까운 구역찾기: find_vma()

find_vma()함수는 프로쎄스기억기서술자의 주소 mm과 선형주소 addr이라는 두개의 변수를 받는다.

이 함수는 vm_end 마당이 addr보다 큰 첫번째 기억기구역을 찾아서 해당 구역의 서술자주소를 반환한다. 해당 구역이 존재하지 않으면 NULL지적자를 반환한다. addr 주소가 모든 기억기구역의 밖에 있을수도 있으므로 find_vma()가 선택한 구역은 반드시 addr을 포함할 필요는 없다는데 류의해야 한다.

매 기억기서술자에는 프로쎄스가 마지막으로 참조한 구역의 서술자주소를 저장하는

mmap_cache마당이 있다. 이 마당은 지정한 선형주소를 포함하는 구역을 찾는데 걸리는 시간을 줄이려고 추가하였다. 프로그람에서 참조하는 주소의 린접성때문에 다음에 참조하는 주소가 들어있는 령역이 마지막으로 참조한 주소가 들어있던 령역과 같을 가능성이 매우 높다.

따라서 맨 먼저 mmap_cache가 가리키는 구역에 addr이 들어가는가를 검사하고 들어가면 해당 구역서술자의 지적자를 반환한다.

```
vma = mm->mmap oache;
if (vma && vma->vm_end > addr && vma->vm_start <= addr)
return vma;
```

그렇지 않으면 프로쎄스의 기억기구역을 검색해야 한다.이 함수는 빨간색-검은색나무에서 기억기구역을 검색한다.

```
rb_node = mm->mm_rb.rb_node;
vma = NULL
while (rb_node) (
   vma_tmp = rb_entry(rb_node, struct vm_area_struct, vm_rb) ;
   if (vma_tmp->vm_end > addr) {
           vrna = vma_tmp;
           if (vma_tmp->vm_ start <= addr);
                   break;
           rb_node = rb_node->rb_ left;
   } else
           rb_node = rb_node->rb_right;
}
if (vma)
   mrn->mmap_ cache = vma;
return vna;
```

이 함수는 빨간색-검은색나무의 마디를 가리키는 지적자에서 해당 기억기구역서술자의 주소를 알아내는 rb_entry마크로를 사용한다. 핵심부는 find_vma_prev()와 find_vma_prepare()함수도 정의한다. (mm\mmap.c)

find_vma_prev()는 변수로 지정한 선형주소 앞에 있는 기억기구역과 다음에 있는 기억기구역의 서술자주소를 반환한다. find_vma_prepare()는 빨간색-검은색나무에서 지정한 선형주소에 해당하는 새로운 마디를 추가할 위치를 찾아서 이전에 있는 기억기구역의 주소와 삽입할 잎(leaf)의 부모마디주소를 반환한다.

(2) 지정한 구간과 겹치는 기억기구역찾기: find_vma_intersection()

find_vma_intersection()함수는 지정한 선형주소구간과 겹치는 첫번째 기억기구역

을 찾는다.(include\linux\mm.h)

mm 변수는 프로쎄스의 기억기서술자를 가리키고 선형주소 start_addr와 end_addr은 구간을 지정한다.

```
vma = find_vma(mm, start_addr) ;
if (vma && end_addr <= vma->vm_ start)
   vna = NULI;
return vma;
```

해당 구역이 존재하지 않으면 NULL을 반환한다. 정확히 말해서 find_vma()가 유효한 주소를 반환하더라도 찾은 기억기구역이 선형주소구간뒤에서 시작한다면 vma를 NULL로 설정한다.

(3) 빈 주소구간찾기: arch_get_unmapped_area()

arch_get_unmapped_area()함수는 프로쎄스의 주소공간을 검색하여 사용가능한 선형주소구간을 찾는다.(mm\mmap.c)

변수 len은 구간길이를 지정하고 변수 addr로 검색을 시작할 주소를 지정할수 있다. 이 함수는 검색이 성공하면 새 구간의 시작주소를 반환하고 실패하면 오유코드 -ENOMEM을 반환한다.

먼저 구간길이가 일반적으로 3GB인 사용자방식선형주소의 제한범위안에 있는가를 확인한다. addr가 0이 아니면 addr부터 시작하는 구간을 할당하려고 시도한다. 안전하게 하려고 addr의 값을 4kB의 배수가 되도록 반올림한다. addr가 0이거나 이전 검색이 실패한 경우 사용자방식선형주소공간의 1/3지점부터 검색을 시작한다.

addr부터 시작하여 addr값을 증가시키며 반복해서 find_vma()를 실행하여 요청한 빈 구간을 찾는다. 검색하는 과정에서 다음과 같은 경우가 있을수 있다.

- 요청한 구간이 선형주소공간에서 아직 검색하지 않은 부분보다 크다.(addr+len>TASK_SIZE) 요청을 처리할만한 충분한 선형주소가 없으므로 -ENOMEM을 반환한다.
- 마지막으로 검색한 구역뒤에 있는 빈 공간의 크기가 충분하지 않다.(vma !=NULL && vma->vm_start< addr+len) 다음 구역으로 넘어간다.
- 우의 두가지 경우에 모두 해당하지 않으면 충분히 큰 빈공간을 찾은것이다. 이때에는 addr를 반환한다.

기억기서술자목록에 구역삽입: insert_vm_struct()

insert_vm_struct()는 기억기구역 객체목록과 빨간색-검은색나무에 vm_area_struct구조체를 삽입한다.

int insert_vm_struct(struct mm_struct * mm, struct vm_area_struct * vma)

```
{
    struct vm_area_struct * __vma, * prev;
    struct rb_node ** rb_link, * rb_parent;

    if (!vma->vm_file) {
            BUG_ON(vma->anon_vma);
            vma->vm_pgoff = vma->vm_start >> PAGE_SHIFT;
    }
    __vma = find_vma_prepare(mm, vma->vm_start, &prev, &rb_link,
&rb_parent);
    if (__vma && __vma->vm_start < vma->vm_end)
            return -ENOMEM;
    vma_link(mm, vma, prev, rb_link, rb_parent);
    return 0;
}
```

이 함수는 프로쎄스기억기서술자의 주소를 가리키는 mm과 삽입할 vm_area_struct 객체의 주소를 가리키는 vma라는 두 변수를 받는다. 기억기구역객체의 vm_start와 vm_end마당은 이미 초기화되여있어야 한다. 이 함수는 find_vma_prepare()함수를 호출하여 빨간색-검은색나무인 mm->mm_rb에서 vma를 넣을 위치를 찾는다. 다음으로 insert_vm_struct()는 vma_link()함수를 호출한다. (mm\mmap.c)

이 함수는 다음과 같은 기능을 수행한다.

1. mm->page_table_lock 스핀잠그기를 획득한다.
2. rnm->mmap가 가리키는 련결목록에 기억기구역을 삽입한다.
3. mm->mm_rb 빨간색-검은색나무에 기억기구역을 삽입한다.
4. mm->page_table_lock 스핀잠그기를 해제한다.
5. mm->map_count계수기를 하나 증가시킨다.

해당 구역에 파일에 대한 기억기배치가 들어있으면 추가적인 작업을 진행한다.

핵심부는 insert_vm_struct()와 똑같지만 mm이 참조하는 기억기구역자료구조를 수정하기 전에 어떤 종류의 잠그기도 획득하지 않는 __insert_vm_start()함수도 정의한다. 핵심부는 이미 해당하는 잠그기를 획득한 경우처럼 기억기구역자료구조를 동시에 접근하는 일이 발생할수 없다고 확신할 때 이 함수를 사용한다.

__vm_unlink()함수는 기억기서술자주소 mm과 두 기억기구역객체주소인 vma와 prev를 변수로 받는다. vma와 prev기억기구역은 모두 mm에 속해야 하며 기억기구역순서에서 prev는 vma앞에 있어야 한다. 이 함수는 기억기서술자의 련결목록과 빨간색-검은색나무에서 vma를 제거한다.

(4) 선형주소구간할당

여기서는 새로운 선형주소구간을 할당하는 방법에 대하여 보기로 하자. 이 작업을 수행하기 위해 do_mmap()함수를 리용하여 current프로쎄스용으로 새 기억기구역을 생성하고 초기화한다.(INCLUDE\linux\mm.h) 성공적으로 할당한 후 이 기억기구역을 프로쎄스의 다른 기억기구역과 합칠수 있다.

do_mmap()함수는 다음과 같은 변수를 사용한다.

file과 offset

새 기억기구역이 파일을 기억기로 배치하는 경우에 파일서술자지적자 file과 파일편위 offset를 사용한다. 여기서는 기억기배치가 필요하지 않으며 file과 offset가 모두 NULL이라고 가정한다.

addr

빈 구간을 찾기 시작할 선형주소를 지정한다.

len

선형주소구간 길이이다.

prot

기억기구역에 들어가는 페지의 접근권한을 지정한다. 사용할수 있는 기발은 PROT_RFAD의 PROT_WRITE, PROT_EXEC, PROT_NONE이다. 처음 세 기발은 VM_READ와 VM_WMTE, VM_EXEC와 의미가 같다. PROT_NONE은 프로쎄스가 이런 접근권한을 하나도 소유하지 않음을 의미한다.

flag

기억기구역의 나머지기발을 지정한다.

MAP_GROWSDOWN, MAP_LOCKED, MAP_DENYWRITE, MAP_EXECUTABLE

표 3-4에서 보여준 기발과 의미가 같다.

MAP_SHARED, MAP_PRIVATE

MAP_SHARED는 기억기구역에 있는 페지를 여러 프로쎄스가 공유할수 있음을 나타내고 MAP_PRIVATE는 이와 정반대의 의미이다. 두 기발은 vm_area_strcut서술자의 VM_SHARED기발을 참조한다.

MAP_ANONYMOUS

기억기구역에 련관된 파일이 없다.

MAP_FIXED

구간의 시작선형주소는 addr변수로 지정한것이여야 한다.

MAP_NORESERVE

여유 페지들의 수를 미리 검사할 필요가 없다.

do_mmap()함수는 먼저 offset값에 대해 몇가지 기초적인 검사를 수행한 후 do_mmap_pgoff()함수를 실행한다.(MM\mmap.c)

새로운 선형주소구간이 디스크에 있는 파일을 배치하는것이 아니라면 do_mmap_pgoff()함수는 다음과 같은 단계를 수행한다.

1. 변수의 값이 정확한지, 요청을 처리할수 있는지를 검사한다. 특히 요청을 처리하지 못하게 만드는 다음 조건에 해당하는지 검사한다.

- 선형주소구간의 길이가 0이거나 TASK_SIZE보다 큰 주소를 포함한다.
- 프로세스가 너무 많은 기억기구역을 배치하고있어서 mm기억기서술자의 map_count마당값이 MAX_MAP_COUNT값을 초과한다.
- flag변수가 새로운 선형주소구간의 페지를 RAM에서 잠그도록(교환하여 내보내지 못하도록) 지시하는데 프로세스가 잠그는 페지의 수가 프로세스서술자의 rlim[LIMIT_MEMLOCK].rlim_cut 마당에 들어있는 한계를 초과한다.

이 조건들가운데서 하나라도 해당하면 do_mmap_pgoff()는 부값을 반환하며 완료한다. 선형주소구간의 길이가 0이면 아무 일도 하지 않고 완료한다.

2. 새로운 구역을 위한 선형주소구간을 얻는다. MAP_FIXED기발을 지정한 경우 addr값에 몇가지 검사를 수행하고 그렇지 않으면 arch_get_unmapped_area()함수를 호출하여 구간을 획득한다.

```
if (flags & MAP_FIXED) {
        if (addr + len > TASK_SIZE)
                return -ENOMEM;
        if (addr & ~PAGE_MASK)
                return -EINVAL;
} else
        addr = arch_get_unmapped_area (file, addr, len, pgoff, flags);
```

3. prot와 flags변수가 저장하는 값을 결합하여 새 기억기구역의 기발을 계산한다.(calc_vm_flags()함수는MM\nommu.c에 정의되여있다.)

```
vm_flags = calc_vm_flags (prot, flags) | mm->def_flags
        | VM_MAYREAD | VM_MAYWRITE | VM_MAYEXEC;
if (flags & MAP_SHARED)
        vm flags |= VM_ SHARED | VM_MAYSHARE;
```

calc_vm_flags()함수는 prot에 PROT_READ, PROT_WRITE, PROT_EXEC기발이 설정되여있는 경우에만 vm_flags의 VM_READ, VM_WRITE, VM_EXEC기발을 설정한다. 또한 flags에 MAP_GROWSDOWN, MAP_DENWRITE, MAP_EXECUTABLE기발이 설정되여있는 경우에만 vm_flags의 VM_GROWSDOWN, VM_DENYWRITE, VM_EXECUTABLE기발을 설정한다. vm_flags의 VM_MAYREAD, VM_MAYWRITE, VM_MAYEXEC기발과 모든 기억기구역의 기본기발인 mm->dev_flags에서 설정한 기발을 각각 1로 설정하고

기억기구역을 다른 프로쎄스와 공유해야 한다면 VM_SHARED와 VM_MAYSHARE기발도 1로 설정한다.

4. find_vma_prepare()를 호출하여 새 구간앞에 있을 기억기구역객체와 빨간색-검은색나무에서 새 구역의 위치를 찾는다.

```
for (; ; ) {
        vma =
          find_vma_prepare(mm, addr, &prev, &rb_1ink, &rb_parent) ;
        if ( !vma || vma->vm_start >= addr + len)
  break;
        if (do_munmap(mm, addr, len) )
  return -ENOMEM;
}
```

find_vma_prepare()함수는 새 구간과 겹치는 기억기구역이 이미 존재하는가도 검사한다. 이 경우 이 함수는 새 구간이 끝나기 전에 시작하는 기억기구역을 가리키는 NULL이 아닌 주소를 반환한다. 이 경우 do_mmap_pgoff()함수는 do_munmap()을 호출하여 새 구간을 제거한 후 전체 과정을 다시 시작한다.(뒤에 나오는 《선형주소구간해제》를 참고)

5. 새로 기억기구간을 추가함으로써 페지주소공간의 크기인 mm->total_vm<<PAGE_SHIFT+len이 프로쎄스서술자의 rlim[RLIMIT_AS].rlim_cur 마당이 지정하는 한계를 넘는가를 검사한다. 넘는다면 오유코드 -ENOMEM을 반환한다. 이 검사를 1단계에서 하지 않고 여기서 하는 리유는 4단계에서 다른 기억기구역을 지웠을 수도 있기때문이다.

6. flags변수가 MAP_NORESERVE기발을 지정하지 않으면서 새로운 기억기구간이 쓰기가능한 개인페지를 포함해야 하며 여유페지들의 수가 선형주소구간의 크기(페지단위)보다 작으면 오유코드 -ENOMEM을 반환한다. 이 마지막 검사는 vm_enough_memory()함수가 수행한다.

7. 새 구간이 개인용이면서(VM_SHARED를 지정하지 않음) 디스크에 있는 파일을 배치하지 않는다면 vma_merge()를 호출하여 이전 기억기구역이 새 구간을 포함하도록 확장할수 있는지 검사한다. 물론 이전 기억기구역은 vm_flags 변수에 저장하고 있는 기발과 완전히 똑같은 기발을 소유해야 한다. 이전 기억기구역을 확장할수 있다면 vma_merge()는 다음에 있는 기억기구역과도 합치려고 시도한다.(이것은 새 구간이 두 기억기구간사이의 구멍(hole)을 메우면서 세 구역이 모두 같은 기발을 소유할 때이다.) 이전 기억기구간을 확장하는데 성공하면 12단계로 건너뛴다.

8. kmem_cache_alloc()스랩(slab)할당자함수를 호출하여 새 기억기구역용으로 vm_area_struct자료구조를 할당한다.

9. 새 기억기구역객체(vma가 가리키는)를 초기화한다.

```
vma->vm_mm = mm;
vma->vrn_start = addr;
vma->vm_end = addr + len;
vma->vm_flags = vm_flags;
vma->vm_page_prot = protection_map[vm_flags & OxOf] ;
// vma->vm_ops = NULL;
vma->vm_pgoff = pgoff;
// vma->vm_file = NULL;
// vma->vm_private_data = NULL;
// vma->vm_raend = 0;
```

만일 VM_GROWSDOWN|VM_GROWSUP기발이 설정되여있지 않고 다음 VM_DENYWRITE기발이 설정되여있는 경우 deny_write_access()함수를 호출하여 쓰기가능한 경우 vma->vm_file = file; 를 초기화 한다.

deny_write_access()함수는 현재 파일구조체의 inode를 얻어 쓰기참조계수를 평가함으로써 이 파일을 쓸수 있는가를 검사한다.

10. MAP_SHARED(VM_SHARED) 기발을 지정하고있으면(그리고 새 기억기구간이 디스크에 있는 파일을 배치하지 않으면) 이 구역을 IPC공유기억기로 사용하는것이다. shmem_zero_setup()을 호출하여 이것을 초기화한다.

11. vma_link()를 호출하여 새 구역을 기억기구역목록과 빨간색-검은색나무에 추가한다(앞서 본《기억기서술자목록에 구역삽입: insert_vm_struct()》를 참고하시오.)

12. 기억기서술자의 total_vm마당에 들어있는 프로쎄스주소공간의 크기를 증가한다.

```
mm->total_vm += len >> PAGE_SHIFT;
__vm_stat_account(mm, vm_flags, file, len >> PAGE_SHIFT);
```

13. VM_LOCKED기발을 지정하고있으면 mm->locked_vm에 있는 잠근 페지의 개수를 증가시킨다.

```
if (vm_flags & VM_LOCKED) {
        mm->locked_vm += len >> PAGE_SHIFT;
        make_pages_present(addr, addr + len);
}
```

make_pages_present()함수를 호출하여 기억기구역에 있는 모든 페지를 련속해서 할당하고 해당 페지를 RAM에서 잠근다. make_pages_present()함수는 아래와 같이 get_user_pages()를 호출한다.

get_user_pages()함수는 addr와 addr+len 사이에 있는 모든 페지의 시작하는 선형주소마다 각각 follows_page()를 반복해서 호출하여 current의 페지표에

물리페지에 대한 배치가 있는지 검사한다.

그런 물리페지가 존재하지 않으면 get_user_pages()는 handle_mm_fault()를 호출한다.

《주소공간안의 잘못된 주소처리하기》에서 보지만 handle_mm_fault()는 한 페지틀을 할당하고 기억기구역서술자의 vm_flags마당에 따라 페지표입구점을 설정한다.

14. MAP_POPULATE기발이 지정되여있다면

```
        if (flags & MAP_POPULATE) {
                up_write(&mm->mmap_sem);
                sys_remap_file_pages(addr, len, 0,
                                pgoff, flags & MAP_NONBLOCK);
        down_write(&mm->mmap_sem);
        }
```

up_write()함수와 down_write()함수는 씨매포에 대한 쓰기잠그기를 해방 또는 잠그는 함수로써 쓰기한 후에 잠그기를 해방하며 쓰기에 대해 잠근다.

```
 /*
 * 쓰기후 잠그기를 해방한다.
 */
static inline void up_write(struct rw_semaphore *sem)
{
   rwsemtrace(sem,"Entering up_write");
   __up_write(sem);
   rwsemtrace(sem,"Leaving up_write");
}

/*
 * 쓰기에 대하여 잠근다.
 */
static inline void down_write(struct rw_semaphore *sem)
{
   might_sleep();
   rwsemtrace(sem,"Entering down_write");
   __down_write(sem);
   rwsemtrace(sem,"Leaving down_write");
}
```

sys_remap_file_pages()는 MM\fremap.c에 정의되여있는데 새 기억기구역의 선형주소를 얻는다.

15. 마지막으로 새 기억기구역의 선형주소를 반환하며 완료한다.

(5) 선형주소구간의 해제

do_munmap()함수는 현재프로쎄스의 주소공간에서 선형주소구간을 제거한다. 이 함수는 구간의 시작주소인 addr와 구간의 길이인 len을 변수로 사용한다. 대체로 지우려는 구간은 기억기구역 하나에만 해당하지 않는다. 해당 구간이 한 기억기구역에 들어가기도 하지만 두 구역이상에 걸쳐 있을수도 있다. (MM\nommu.c)

이 함수는 크게 두 단계를 거친다. 1단계에서는 프로쎄스소유의 기억기구역목록을 검색하여 선형주소구간과 겹치는 모든 구역을 지운다. 2단계에서는 프로쎄스페지표를 갱신하고 1단계에서 제거한 기억기구역의 크기를 줄인 구역을 다시 삽입한다.

1단계: 기억기구역검색

do_mummap()함수는 다음 단계를 수행한다.

1. 본격적인 작업에 앞서 변수값을 검사한다. 선형주소구간이 TASK_SIZE보다 큰 주소를 포함하거나 addr가 4096의 배수가 아니거나 선형주소구간의 길이가 0이면 오유코드 -EINVAL을 반환한다.

2. 삭제할 선형주소구간과 겹치는 첫번째 기억기구역을 찾는다.

```
mpnt = find_vma_prev(current->mm, addr, &prev);
if ( !mpnt || mpnt->vm_start >= addr + len)
        return 0;
```

3. 선형주소구간이 기억기구역중간에 들어있을 때 이것을 삭제하면 구역이 작은 구역 두개로 나누어질것이다.

split_vma()함수로 기억구역을 쪼갠다.

4.해방된 vma목록을 만들고 mm의 vma목록으로부터 삭제하며 실제페지를 해방한다. 다음 페지표를 호출하여 지적된 령역에서 페지표정보의 레코드ID를 얻는다. 이때 대용량페지인경우와 일반페지인 경우를 갈라 처리한다.

5.선형주소구간과 겹치는 모든 기억기구역의 서술자를 목록으로 만든다. 기억기구역서술자의 vm_next 마당이 목록앞에 들어있는 요소를 가리키게 하여 목록을 만들수 있다.(따라서 이 마당은 이전것을 가리키는 역할을 한다.) 각 구역을 이 목록에 추가할 때마다 변수 free가 마지막으로 추가한 요소를 가리키게 한다. 이 목록에 추가한 구역을 프로쎄스소유의 기억기구역목록과 빨간색-검은색나무에서 제거한다.(rb_erase()함수를 리용한다.)

```
npp = (prev ? &prev->vm_next : &current->mm->mmap) ;
free = NULL;
spin_lock (&current->rnm->page_table_lock) ;
```

```
for (; mpnt && mpnt->vm_start < addr + len; mpnt = *npp) {
        *npp = mpnt->vm_next;
        mpnt->vm_next =  free;
        free = mpnt;
        rb_erase(&mpnt->vm_rb, &current->mm->mm_rb);
}
current->mrn->mmap_cache = NULL;
spin_unlock(&current->mm->page_table_lock) ;
```

2단계: 페지표갱신

free가 가리키는 기억기구역서술자부터 시작하여 1단계에서 만든 기억기구역 목록을 살펴보려고 while문을 사용한다.

반복때마다 변수 mpnt는 목록에 있는 기억기구역의 서술자를 가리킨다.

current->mm기억기서술자의 map_count 마당값을 1씩 줄이고(1단계에서 프로쎄스소유의 기억기구역목록에서 구역을 이미 삭제했기때문에) mpnt구역을 제거해야 하는지 아니면 단순히 크기를 줄여야 하는지 결정하기 위한 검사를 한다.(이것은 두 의문형 조건문에서 한다.)

```
current ->nm->map_count--;
st = addr < mpnt->vrn_start ? mpnt->vm_start : addr;
end = addr+len;
end = end > mpnt->vm_end ? mpnt->vm_end : end;
size = end - st;
```

변수 st와 end는 mpnt가 가리키는 기억기구역에서 삭제할 선형주소구간의 경계를 지정한다. 변수 size는 구간의 길이를 지정한다.

다음으로 do_munmap()은 st와 end 사이구간에 들어있는 페지에 할당한 페지틀을 해제한다.

```
zap_page_range (mm, st, size) ;
```

zap_page_range()함수는 st와 end 사이의 구간에 들어있는 페지틀을 해제하고 해당 페지표입구점을 갱신한다. 이 함수는 페지표를 조사하려고 zap_pmd_range()와 zap_pte_range()함수를 중첩해서 호출한다. zap_pte_range()함수가 페지입구점표를 지우고 해당 페지틀을 해제한다.(또는 교체기억기령역으로 보낸다.) 이 과정에서 zap_pte_range()는 st부터 end 사이의 구간에 해당하는 TLB 입구점을 비우는 일도 한다.

do_munmap()순환에서 반복을 할 때마다 마지막으로 하는 일은 mpnt가 가리키는 기억기구역을 축소하여 current의 기억기구역목록에 다시 삽입해야 하는지 검사하는것이다.

```
extra = unmap_fixup(mm, mpnt, st, size, extra);
```

unmap_fixup()함수는 다음 4가지 가능한 경우를 고려한다.

1. 기억기구역이 완전히 없어졌다. 이전에 할당한 기억기구역객체의 주소를 반환한다.(《1단계: 기억기구역 검색》에서 4단계참고) kmem_cache_free()를 호출하여 이 객체를 해제할수 있다.

2.기억기구역의 뒤부분만 삭제되였다.

(mpnt->vm_start < st) && (mpnt->vm_end == end)

이 경우 mpnt의 vm_end 마당과 갱신하고 __insert_vm-_truct()를 호출하여 프로쎄스소유의 기억기구역목록에 축소된 구역을 삽입한 후 이전에 할당한 기억기구역객체의 주소를 반환한다.

3.기억기구역의 앞부분만 삭제되였다.

(mpnt->vm_start == st) && (mpnt->vm_end > end)

이 경우 mpnt의 vm_start 마당과 갱신하고 __insert_vm_struct()를 호출하여 프로쎄스소유의 기억기구역목록에 축소된 구역을 삽입한 후 이전에 할당한 기억기구역객체의 주소를 반환한다.

4. 선형주소구간이 기억기구역의 중간에 있다.

(mpnt->vm_start < st) && (mpnt->vm_end > end)

mnpt와 이전에 할당한 여분의 기억기구역서술자의 vm_start와 vm_end 마당과 갱신하여 이것들이 각각 mpnt->vm_start에서 st사이의 구간과 end에서 mpnt->vm_end 사이의 구간을 가리키게 한다. 그후 __insert_vm_struct()를 두번 호출해서 두 구역을 프로쎄스소유의 기억기구역목록과 빨간색-검은색나무에 추가한 후 NULL을 반환해서 이전에 할당한 여분의 기억기구역서술자를 보존한다.

이것으로 do_munmap()의 2단계 순환에서 한번 반복할 때 수행하는 일에 관한 설명을 마쳤다.

1단계에서 만든 목록에 들어있는 모든 기억기구역서술자를 처리한 후 do_munmap( )은 여분의 기억기서술자를 사용했는지 여부를 검사한다. unmap_fixup()이 NULL을 반환하면 이 서술자를 사용한것이고 그렇지 않으면 do_munmap()는 kmem_cache_free()를 호출하여 이것을 해제한다. 마지막으로 do_munmap()는 free_pgtabIe()함수를 호출한다. 이 함수는 방금 제거한 선형주소구간에 해당하는 페지표입구점을 검색하여 사용하지 않는 페지표를 저장하는 페지틀을 회수한다.

4. 페지오유례외조종기

앞에서 언급한것처럼 Linux의 페지오유례외조종기는 프로그람작성오유때문에 발생한 례외와 정당하게 프로쎄스주소공간에 들어있지만 아직 할당하지 않은 페지를 참조해서 발생한 례외를 구별해야 한다.

기억기구역서술자는 례외조종기가 이 일을 효과적으로 수행할수 있게 한다. 80x86 구조용 페지오유례외 조종기인 do_page_fault()함수는 페지오유가 발생한 선형주소와

current프로쎄스의 기억기구역을 비교한다.

그래서 그림 3-10과 같이 례외를 처리하는 옳바른 방법을 결정한다.

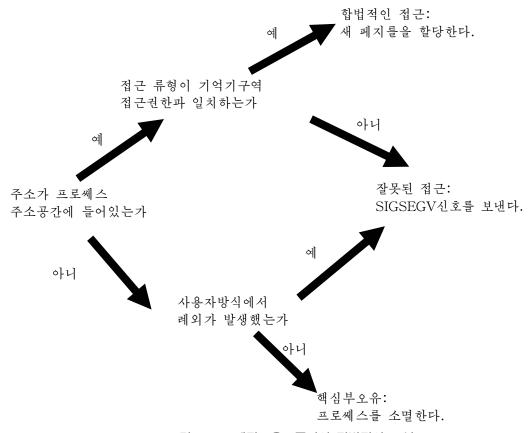



그림 3-10. 페지오유조종기의 전반적인 구성

페지오유조종기는 전체적인 구조에 잘 맞지 않는 몇가지 특이한 경우를 구별해야 하고 여러 가지 정당한 접근을 구별해야 하기때문에 실제작업은 그림에 나타낸것보다 훨씬 복잡하다. 그림 3-11은 조종기의 자세한 순서도이다.

do_page_fault()에 나오는 vmalloc_fault와 good_area, bad_area, no_context 표식은 순서도블로크와 코드의 특정행을 련관시키는데 도움을 준다.

do_page_fault()함수는 다음과 같은 입력변수를 받는다.

· 례외가 발생할 당시의 극소형처리기 등록기값을 포함하는 pt_regs 구조체의 주소 regs

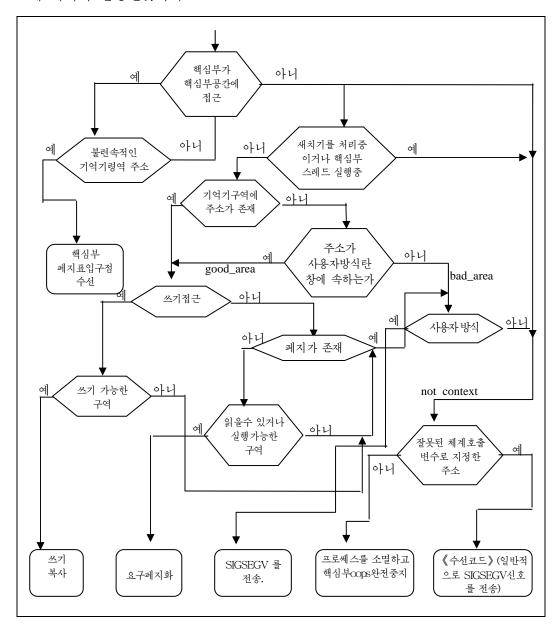
· 례외가 발생할 때 조종장치가 탄창에 저장하는 3bit error_code. 이 비트의 의미는 다음과 같다.

– 비트0이 0이면 존재하지 않는 페지에 접근할 때 례외가 발생한것이다. (페

지표 입구점에 있는 Present기발이 0이다.) 비트 0이 1이면 잘못된 접근권한때문에 례외가 발생한것이다.

- 비트 1이 0이면 읽기나 실행접근, 1이면 쓰기접근때문에 례외가 발생한것이다.
- 비트 2가 0이면 처리기가 핵심부방식에 있을 때, 1이면 사용자방식에 있을 때 례외가 발생한것이다.

그림 3-11. 페지오유조종기의 순서도

do_page_fault()가 하는 첫번째 작업은 페지오유가 발생한 선형주소를 읽는것이다. 례외가 발생할 때 CPU조종장치는 이 값을 cr2조종등록기에 저장한다.

```
__asm__("movl %%cr2,%0":"=r" (address));
if (notify_die(DIE_PAGE_FAULT, "page fault", regs, error_code, 14,
                                    SIGSEGV) == NOTIFY_STOP)
    return;
if (regs->eflags & (X86_EFLAGS_IF|VM_MASK))
    local_irq_enable();
tsk = current;
```

오유가 발생한 선형주소를 변수 address에 저장한다. 또한 오유가 발생하기 전에 새치기를 허용하고있었으면 이것을 허용한다. current프로쎄스서술자 지적자를 변수 tsk에 저장한다.

그림 3-11의 웃부분에서 보는바와 같이 do_page_fault()는 먼저 잘못된 선형주소가 마지막 1GB령역에 들어있는지 그리고 핵심부가 존재하지 않는 페지틀을 접근하다가 례외가 발생한것인지 검사한다.

```
if (unlikely(address >= TASK_SIZE)) {
    if (!(error_code & 5))
            goto vmalloc_fault;
```

만일 CONFIG_X86_4G인 경우 처리는 다음과 같다.

```
#ifdef CONFIG_X86_4G
    /* If it's vm86 fall through */
    if (unlikely(!(regs->eflags & VM_MASK) && ((regs->xcs & 3) == 0)))
{
            if (error_code & 3)
                    goto bad_area_nosemaphore;
            goto vmalloc_fault;
    }
#else
    … …
```

vmalloc_fault표식에 있는 코드는 핵심부방식에서 불련속적인 기억기령역을 접근하다가 발생한것과 같은 오유를 처리한다. 이 경우에 대해서는 후에 보게 되는 《불련속적인 기억기령역접근》에서 설명한다. 다음으로 례외가 새치기를 처리하거나 핵심부스레드를 실행하는 도중에 발생했는지 검사한다. 핵심부스레드의 프로쎄스서술자의 mm마당은 항상 NULL이라는 사실을 기억할것이다.

```
info.i_code = SEGV_MAPERR;
```

```
if (in_interrupt( ) || !tsk->mm)
   goto no_context;
```

두 경우 모두 do_page_fault()는 선형주소를 current의 기억기구역과 비교하지 않는다. 새치기조종기와 핵심부스레드는 TASK_SIZE아래의 선형주소를 전혀 사용하지 않으므로 기억기구역을 전혀 사용하지 않는다. 따라서 이것을 비교하는것은 의미가 없다.(변수 info에 대한 정보와 no_context표식에 있는 코드에 관한 설명은 다음절에서 볼수 있다.)

페지오유가 새치기조종기나 핵심부스레드에서 발생하지 않았다고 하자. 그러면 이 함수는 프로쎄스소유의 기억기구역을 조사하여 오유가 발생한 선형주소가 프로쎄스의 주소공간에 들어있는가를 검사한다.

```
if (!down_read_trylock(&mm->mmap_sem)) {
          if ((error_code & 4) == 0 &&
              !search_exception_tables(regs->eip))
                 goto bad_area_nosemaphore;
          down_read(&mm->mmap_sem);
   }
   vma = find_vma(mm, address);
   if (!vma)
          goto bad_area;
```

vma가 NULL이면 address이후에 끝나는 기억기구역이 없다는 의미이다. 따라서 오유가 발생한 주소는 분명히 잘못된 주소이다. 반면에 address이후에 끝나는 첫번째 기억기구역이 address를 포함하는 경우 good_area 표식에 있는 코드로 뛰여넘는다.

이 두 가지 if조건에 모두 해당하지 않으면 address를 포함하는 기억기구역이 없다는 의미이다. 그러나 오유가 발생한 주소가 프로쎄스의 사용자방식탄창에 push나 pusha명령을 사용했기때문일수도 있으므로 추가로 이것을 검사해야 한다.

여기서 탄창을 기억기구역에 배치하는 방법을 보기로 하자. 탄창이 들어있는 구역을 낮은 주소쪽으로 확장한다. 이 구역의 VM_GROWSDOWN기발을 설정하므로 vm_end마당값은 바뀌지 않는 반면에 vm_start마당의 값은 감소할수 있다. 이 구역의 범위는 사용자방식탄창의 현재 크기를 포함하지만 정확하게 경계를 구분하지 않는다. 이런 요인이 생기는 리유는 다음과 같다.

- 탄창크기는 임의인 반면에 구역의 크기는 4kB의 배수다.(구역은 완전한 페지를 포함해야 한다.)• 구역에 할당한 페지들은 해당구역을 없애기전까지 절대로 해제되지 않는다. 특히 탄창이 들어있는 구역의 vm_start마당값은 감소만 할수 있으며 절대로 증가하지 않는다. 프로쎄스가 일련의 pop명령을 수행하여 탄창지적자가 증가하더라도 구역의 크기는 변하지 않고 그대로이다.

이제는 탄창용으로 할당한 마지막 페지를까지 모두 사용한 프로쎄스가 어떻게 페지오유레외를 일으킬수 있는가는 명확해졌을것이다. 이것은 push명령을 리용해서 구역밖에 있는 주소를(그리고 존재하지 않는 페지들을) 참조하는 경우이다. 이런 종류의 레외는 프로그람작성을 잘못하여 발생한것이 아니므로 페지오유조종기는 이것을 별도로 처리해야 한다.

다시 do_page_fault()로 돌아가 방금전에 설명한 경우를 검사하는 코드를 살펴보자.

```
if (!(vma->vm_flags & VM_GROWSDOWN))
   goto bad_area;
if (error_code & 4     /* 사용자방식 */
   && address + 32 < regs->esp)
   gotobad area;
if (expand_stack(vma, address))
   goto bad_area;
```

goto good-area; 구역의 기발에 VM_GROWSDOWN기발이 설정되여있으면서 사용자방식에서 레외가 발생한 경우 address가 탄창지적자 regs->esp보다 작은가를 검사한다.(조금만 작아야 한다.) 탄창과 관련한 몇가지 기호언어명령(pusha 같은)은 기억기접근후에만 esp등록기를 감소시키기때문에 프로쎄스에 32B의 허용구간을 준다. 주소가 충분히 높으면(허용구간내에서) expand_stack()함수를 호출하여 프로쎄스가 탄창과 주소공간을 확장할수 있는가를 검사한다. expand_stack()은 모든것이 이상없으면 vma의 vm_start마당을 address로 설정한 후 0을 반환하고 그렇지 않으면 -ENOMEM을 반환한다.

앞서 살펴본 코드는 구역에 VM_GROWSDOWN기발이 설정되여있으면서 사용자방식에서 레외가 발생한것이 아닌 경우 허용구간을 검사하지 않는다. 이 경우는 핵심부가 사용자방식탄창의 주소를 지정한것으로 항상 expand_stack()을 수행해야 한다.

1) 주소공간밖의 잘못된 주소처리하기

address가 프로쎄스의 주소공간에 들어있지 않으면 do_page_fault()는 bad_area 표식에 있는 문장부터 실행한다. 사용자방식에서 오유가 발생하면 현재프로쎄스에 SIGSEGV신호를 보내고 완료한다.

```
bad area:
      up_read(&tsk->mm->mmap_sem) ;
bad_area_nosemaphore:
  if (error_code & 4) {    /* 사용자방식 */
   if (is_prefetch(regs, address, error_code))
         return;
   tsk->thread.cr2 = address;
```

```
    tsk->thread.error_code = error_code | (address >= TASK_SIZE);
    tsk->thread.trap_no= 14;
    info.si_signo= SIGSEGV;
    info.si_errno = 0;
    info.si_addr = (void *) address;
    force_sig_info(SIGSEGV, &info, tsk) ;
    return;
  }
```

force_sig_info()함수는 프로쎄스가 SIGSEGV신호를 무시하거나 차단하지 않는다는 사실을 확인한 후 변수 info에 추가정보를 담아 사용자방식프로쎄스에 신호를 전송한다. info.si_code마당은 이미 SEGV_MAPERR(존재하지 않는 페지틀때문에 례외가 발생한 경우)나 SEGV_ACCERR(존재하는 페지틀에 잘못된 접근을 하여 례외가 발생한 경우)로 설정되여있다.

핵심부방식에서 례외가 발생하였다면(error_code의 비트2의 값이 0이다) 다음 두 경우중 하나이다.

- 체계호출변수로 핵심부에 전달한 선형주소를 사용하는중에 례외가 발생하였다.
- 실제 핵심부오유때문에 례외가 발생하였다.

do_page_fault()함수는 다음과 같이 이러한 두 경우를 구별한다.

```
no context:
if( (fixup = search_exception_table(regs->eip) ) != 0){
    regs->eip = fixup;
    return;
}
```

첫번째 경우에는 《표식코드(fixup code)》로 이행한다. 이 코드는 대개 current에 SIGSEG신호를 보내거나 적당한 오유코드를 반환하며 체계호출조종기를 끝낸다.

두번째 경우에는 CPU등록기와 핵심부방식탄창을 조작탁과 체계통보문완충기로 이행(dump)한 후 do_exit()를 호출해서 현재프로쎄스를 소멸한다. 이것이 《kernel oops》오유로 출력하는 통보문에서 붙은 이름이다. 핵심부해커는 이행한 값을 사용하여 오유가 발생한 상황을 재구성하고 오유를 찾아 고칠수 있다.

2) 주소공간안의 잘못된 주소 처리하기

do_page_fault()는 address가 프로쎄스의 주소공간에 속하면 good_area표식이 있는 문장부터 실행한다.

```
good_area:
    info.si_code = SEGV_ACCERR;
    write = 0;
```

```
    switch (error_code & 3) {
        default:     /* 3: 현재 쓰기 */
#ifdef TEST_VERIFY_AREA
            if (regs->cs == KERNEL_CS)
                printk("WP fault at %08lx\n", regs->eip);
#endif
            /* fall through */
        case 2:          /* 쓰기접근  */
            if (!(vma->vm_flags & VM_WRITE))
                goto bad_area;
            write++;
            break;
        case 1:          /* 읽기 */
            goto bad_area;
        case 0:          /* 읽기접근 */
            if (!(vma->vm_flags & (VM_READ | VM_EXEC)))
                goto bad_area;
    }
```

쓰기접근때문에 례외가 발생하였다면 기억기구역이 쓰기가능한가를 검사한다. 쓰기가 가능하지 않으면 bad_area코드로 이행하고 쓰기가 가능하면 변수 write를 1로 설정한다.

례외가 읽기나 실행접근때문에 발생하였다면 페지가 이미 RAM에 존재하는가를 검사한다. 페지가 존재하면 사용자방식에 있는 프로쎄스가 특권이 필요한 페지를 (User/Supervisor 기발이 0)에 접근하려다가 례외가 발생한것이므로 bad_area로 이행한다. 페지가 존재하지 않으면 기억기구역이 읽기나 실행이 가능한가도 검사한다.

례외를 발생시킨 접근류형과 기억기구역의 접근권한이 일치하면 handle_mm_fault()함수를 호출하여 새로 페지틀을 할당한다.

```
survive:
    switch (handle_mm_fault(mm, vma, address, write)) {
        case VM_FAULT_MINOR:
            tsk->min_flt++;
            break;
        case VM_FAULT_MAJOR:
            tsk->maj_flt++;
            break;
```

```
            case VM_FAULT_SIGBUS:
                    goto do_sigbus;
            case VM_FAULT_OOM:
                    goto out_of_memory;
            default:
                    BUG();
    }
```

handle_mm_fault()함수는 MM\memory.c에 정의되여있다.

handle_mm_fault()함수는 프로쎄스에 새로 페지틀을 성공적으로 할당한 경우 1이나 2를 반환한다. 1은 현재프로쎄스를 차단하지 않고 페지오유를 처리했다는것을 의미하며 이런 종류의 페지오유를 《부오유(minor fault)》라고 부른다. 2는 페지오유에 의해 현재프로쎄스가 잠들었다는것을 의미하며(대부분 디스크에서 자료를 읽어들여서 프로쎄스에 할당한 페지틀을 채우는데 시간을 소비하기때문이다) 이렇게 현재프로쎄스를 차단하는 페지오유를 가리켜 《주오유(major fault)》라고 부른다. 이 함수는 -1(기억기부족)이나 0(그 밖의 오유)을 반환하기도 한다.

handle_mm_fault()가 0을 반환하면 프로쎄스에 SIGBUS신호를 보낸다.

```
if ( !ret) {
    up_read(&tsk->mm->mrnap_sem);
    tsk->thread.cr2 = address;
    tsk->thread.error_code = error_code;
    tsk->thread.trap_no = 14;
    info.si_signo= SIGBUS;
    info.si_errno= 0;
    info.si_code = BUS_ADRERR;
    info.si_addr = (void *) address;
    foroe_sig_info(SIGBUS, &info, tsk) ;
    if ( !(error_code & 4) )  /* 핵심부방식 */
            goto no_context; )
}
```

handle_mm_fault()가 새로 페지틀을 할당하지 못한 경우 핵심부는 보통 현재프로쎄스를 소멸한다. 그러나 current가 init프로쎄스라면 이것을 실행대기렬의 맨끝에 넣고 순서짜기프로그람을 호출한다. init프로쎄스가 실행을 재개하면 handle_mm_fault()가 다시 실행된다.

```
if ( ret == -1) {
    up_read ( &tsk->mm->mmap_sem) ;
```

```
    if (tsk->pid != 1) {
        if (error_code & 4 )  /* 사용자방식 */
            do_exit ( SIGKILL ) ;
        goto no_context ;
    }
    tsk->policy != SCHED_YIELD;
    schedule( );
    down_read(&tsk->mm->mmap_sem);
    goto survive;
}
```

handle_mm_fault( )함수는 다음 4개 변수를 사용한다.

mm

례외가 발생할 때 CPU에서 동작중이던 프로쎄스의 기억기서술자를 가리키는 지적자이다.

vma

례외가 발생한 선형주소를 포함하는 기억기구역의 서술자를 가리키는 지적자이다.

address

례외가 발생한 선형주소이다.

write_access

task가 address에 쓰기를 하려고 시도한 경우 1, 읽기나 실행을 하려한 경우 0으로 설정한다.

이 함수는 맨 먼저 address를 배치하는데 사용하는 페지중간등록부와 페지표가 존재하는가를 검사한다. address가 프로쎄스의 주소공간에 속하더라도 해당 페지표를 아직 할당하지 않았을수도 있으므로 이것을 할당하는 작업이 모든 일에 앞선다.

```
spin_lock(&mm->page_table_lock);
pgd = pgd_offset (nm, address);
pmd = pmd_alloc(mm, pgd, address) ;
if (pmd) {
    pte = pte_alloc (mm, pmd, address);
    if (pte)
return handle_pte_fault (mm, vma, address, write_access, pte);
}
spin_unlock ( &mm->page_table_lock) ;
return - 1;
```

변수 pgd는 address를 참조하는 페지대역등록부입구점을 포함한다. 필요하다면

pmd_alloc()을 호출하여 새로운 페지중간등록부를 할당하고 pte_alloc()을 호출하여 새로운 페지표를 할당한다. 두 작업 모두 성공하면 변수 pte는 address를 참조하는 페지표입구점을 가리킨다. 다음으로 handle_pte_fault()함수를 호출하여 address에 해당하는 페지표를 조사하고 프로쎄스에 새로 페지틀을 어떻게 할당하겠는가를 결정한다.

· 접근한 페지가 존재하지 않으면 즉 해당 페지를 저장하는 페지틀이 없으면 핵심부는 새로운 페지틀을 할당하고 초기화한다. 이런 기법을 《요구페지화(demand paging)》라고 한다.

· 접근한 페지가 존재하지만 읽기전용이면 즉 해당 페지를 저장하는 페지틀이 이미 존재하면 핵심부는 새로운 페지틀을 할당하고 기존 페지틀의 내용을 새로운 프레임으로 복사하여 초기화한다. 이런 기법을 《쓰기복사(Copy On Write)》라고 한다.

3) 요구페지화

요구페지화(demand paging)는 가능한 마지막 순간까지 즉 프로쎄스가 RAM에 존재하지 않는 페지의 주소로 접근하여 페지오유례외가 발생하는 순간까지 페지틀할당을 연기하는 동적기억기할당기법이다.

요구페지화를 시작한 동기는 프로쎄스가 처음부터 자신의 주소공간에 들어있는 모든 주소에 접근하지는 않는다는것이다. 사실 프로쎄스는 이 주소중 일부를 전혀 사용하지 않기도 하고 더 나가서 지역성원리(locality principle)에 따라 프로그람을 실행하는 매 단계에서는 프로쎄스페지의 일부분만 실제로 참조한다. 따라서 일시적으로 사용하지 않는 페지를 담은 페지틀을 다른 프로쎄스가 사용할수도 있다.

요구페지화는 체계에 존재하는 여유페지틀의 평균개수를 늘여 사용가능한 여유기억기를 더 효률적으로 사용할수 있게 하므로 전체를 할당하는것(프로쎄스가 시작하는 즉시 프로쎄스에 모든 페지틀을 할당하고 프로쎄스가 완료할 때까지 기억기에 그대로 유지하는것)보다 훨씬 좋다. 다른 관점에서 보면 요구페지화는 똑같은 량의 RAM으로 더 많은 작업을 처리할수 있게 한다.

핵심부는 요구페지화때문에 발생하는 매 페지오유례외를 처리해야 하므로 CPU박자수를 랑비하게 된다. 다행히 지역성원리에 따라 프로쎄스가 일단 어떤 페지그룹을 리용하여 작업을 시작하면 한동안은 다른 페지를 사용하지 않고 이 페지만 사용한다. 따라서 페지오유례외는 드물게 발생한다고 볼수 있다.

다음과 같은 리유로 접근한 페지가 주기억기에 존재하지 않을수 있다.

· 프로쎄스가 해당 페지에 접근한 사실이 없다. 이 경우 페지표입구점이 0으로 채워져있으므로(pte_none 마크로가 1을 반환한다) 핵심부는 이것을 리용해서 이 경우를 구별할수 있다.

· 프로쎄스가 해당 페지에 접근한 사실이 있지만 페지내용을 일시적으로 디스크에 저장하고있다. 이 경우 페지표입구점이 0으로 채워져있지 않으므로(그러나 페지가 RAM에 존재하지 않으므로 Present기발은 0이다.) 이것을 리용하여 이

경우를 구별할수 있다.

handle_pte_fault()함수는 address가 가리키는 폐지표입구점을 조사하여 두 경우를 구별한다.

폐지를 디스크에 저장하는 경우(do_swap_page()함수를 사용하여)는 《폐지교환하여넣기》에서 살펴보자.

폐지에 접근한 사실이 없다면 do_no_page()함수를 호출한다.(MM\memory.c)

이렇게 빠뜨린 폐지를 읽어들이는 방법은 폐지가 디스크파일을 배치하는가에 따라 두가지가 있다. 이 함수는 기억기구역객체인 vma에 있는 nopage메쏘드를 검사해서 이것을 판단한다. 이 메쏘드는 폐지가 파일을 배치하고있으면 디스크에서 빠뜨린 폐지를 읽어들이는 함수를 가리킨다.

따라서 다음과 같은 경우가 있을수 있다.

- vma->vm_ops->nopage마당이 NULL이 아니다. 이 경우 기억기구역은 디스크파일을 배치하며 이 마당은 폐지를 읽어들이는 함수를 가리킨다. 《기억기배치의 요구폐지화》와 《IPC공유기억기》에서 이 경우를 다룬다.
- vm_ops마당나 vma->vm_ops->nopage마당이 NULL이다. 이 경우 기억기구역은 디스크에 있는 파일을 배치하지 않는다. 즉 《닉명배치(anonymous mapping)》를 한다. 따라서 do_no_page()는 do_anonymous_page()함수를 호출하여 새로운 폐지틀을 할당한다.

```
if ( !vma->vm_ops || !vma->vm_ops->nopage)
        return do_anonymous_page (mm, vma, page table,
                write_acoess, address) ;
```

do_anonymous_page()함수는 쓰기요청과 읽기요청을 별도로 처리한다.

이 함수는 쓰기접근을 할 때에는 alloc_page_vma()를 호출하고 memset마크로를 사용하여 새 폐지틀을 0으로 채운다.

다음으로 tsk의 min_flt마당을 증가시켜 프로쎄스에서 발생한 《부폐지오유(minor page fault)》의 회수를 관리한다. 다음으로 기억기서술자의 rss마당의 값을 증가시켜 프로쎄스에 할당한 폐지틀의 수를 관리한다. 그리고 폐지표입구점에 폐지틀의 물리주소를 설정한다. 이 폐지틀은 쓰기가능하고 내용이 바뀌였다고 표시한다.

lru_cache_add_active()와 mark_page_accessed()함수는 새 폐지틀을 교환과 관련한 자료구조에 추가한다. 이 부분은 뒤에서 설명한다.

반대로 읽기접근을 다룰 때에는 프로쎄스가 해당 폐지에 처음 접근한것이므로 폐지의 내용은 무의미하다. 여기서는 다른 프로쎄스가 써놓은 정보로 채워진 이전 폐지보다는 0으로 채운 폐지를 프로쎄스에 주는것이 더 안전하다.

Linux는 요구폐지화의 기능을 한 단계 더 발전시킨다. 여기서 곧바로 프로쎄스에 0으로 채운 새 프레임을 할당할 필요는 없다. 프로쎄스에 령폐지(zero page)라는 이미

존재하는 페지를 주어 페지틀할당을 연기할수도 있기때문이다. 령페지는 핵심부초기화과정에서 empty_zero_page변수(long integer형을 1024개를 포함하는 배렬로써 0으로 채워져있다)에 정적으로 할당한것이다. 령페지는 5번째 페지틀에 들어있고(물리주소 0x00004000부터 시작한다.) ZERO_PAGE마크로로 참조할수 있다. 따라서 페지표입구점을 령페지의 물리주소로 설정한다.

```
entry = pte_wrprotect (mk_pte (ZERO_PAGE, vma->vm_page_prot))
set_pte (page_table, entry) ;
spin_unlock(&mm->page_table_lock) ;
return 1;
```

페지를 쓰기금지로 표시하기때문에 프로쎄스가 여기에 쓰려고 하면 《쓰기복사》기구가 동작한다. 그리하여 프로쎄스는 이때에만 쓸수 있는 자기만의 페지를 소유하게 된다. 이 기구는 다음에 설명한다.

4) 쓰기복사(Copy On Write)

1세대 Unix체계에서는 서투른 방법으로 프로쎄스생성을 실현하였다. fork()체계호출을 실행하면 핵심부는 말그대로 부모주소공간을 모두 복제하여 자식프로쎄스에 사본을 할당하였다. 이것은 다음과 같은 작업이 필요하기때문에 시간을 많이 소모한다.

- 자식프로쎄스의 페지표용으로 페지를 할당
- 자식프로쎄스의 페지용으로 페지를 할당
- 자식프로프로쎄스의 페지표 초기화
- 부모프로쎄스의 페지를 자식프로쎄스의 해당 페지로 복사

이렇게 주소공간을 생성하는 방법은 많은 기억기접근이 필요하고 CPU박자수를 많이 사용하며 캐쉬의 내용을 완전히 못쓰게 만든다. 마지막으로 더 심각한 문제는 많은 자식프로쎄스가 새로운 프로그람을 적재하고 실행하여 상속받은 주소공간을 완전히 페기하므로 이 작업은 종종 쓸데없는 일이 된다는 점이다.

Linux를 포함한 최근의 Unix핵심부는 《COW(Copy On Write)》라는 좀더 효률적인 접근방법을 사용한다. COW의 개념은 매우 간단하다. 페지틀을 복제하는 대신에 부모프로쎄스와 자식프로쎄스 사이에 페지틀을 공유하는것이다. 그러나 공유하는 동안은 페지틀의 내용을 바꿀수 없다. 부모나 자식프로쎄스가 공유하는 페지틀에 쓰기를 할 때마다 례외가 발생하고 이때 핵심부는 새로운 페지틀으로 페지를 복제하여 이 페지틀을 쓰기가능으로 만든다. 원본페지틀은 쓰기금지상태로 남아서 다른 프로쎄스가 여기에 쓰기를 하려고 시도하면 핵심부는 쓰려는 프로쎄스가 페지틀의 유일한 소유자인지 검사해서 조건에 맞으면 프로쎄스가 페지틀에 쓸수 있게 만든다.

페지서술자의 count마당은 해당 페지틀을 공유하는 프로쎄스의 수를 유지하는데 사용한다. 프로쎄스가 페지틀을 해제하거나 이 페지틀에 쓰기복사를 하게 되면 count마당의 값은 감소한다. 따라서 count가 0이 될 때에만 페지틀을 해제한다. 이제 Linux에

서 COW를 어떻게 실현하는가를 설명한다. handle_pte_fault()함수는 기억기에 존재하는 페지를 접근하다가 페지오유례외가 발생하면 다음 명령을 실행한다.

handle_pte_fault()함수는 방식과 무관계한 함수이다. 이 함수는 페지접근권한과 관련한 가능한 모든 위반을 고려한다. 그러나 80x86방식에서 페지가 접근한다면 해당 접근류형은 쓰기이고 페지틀은 쓰기금지상태이다.(《주소공간안의 잘못된 주소 처리하기》를 참고) 따라서 항상 do_wp_page()함수를 호출한다.

do_wp_page()함수는 페지오유례외와 련관된 페지표입구점이 참조하는 페지틀의 페지서술자를 얻으면서 시작한다. 다음으로 페지를 정말 복제해야 하는가를 결정한다. 한 프로쎄스만 해당 페지를 소유하면 쓰기복사를 적용하지 않고 프로쎄스는 마음대로 해당 페지에 쓰기를 할수 있다. 기본적으로 do_wp_page()함수는 페지서술자의 count마당을 읽어서 값이 1이면 COW를 하면 안된다. 실제로는 페지가 교체캐쉬에 들어갈 때도 count마당이 증가하므로 이 검사는 좀 더 복잡하다. 하여튼 COW를 할 필요가 없으면 다음에 이 페지틀에 쓰기를 하려할 때 더는 페지오유례외가 일어나지 않도록 페지틀을 쓰기가능으로 표시한다.

```
flush_cache_page(vma, address);
entry = maybe_mkwrite(pte_mkyoung(pte_mkdirty(pte)), vma);
ptep_set_access_flags(vma, address, page_table, entry, 1);
update_mmu_cache(vma, address, entry);
pte_unmap(page_table);
spin_unlock(&mm->page_table_lock);
```

COW를 통해 해당 페지를 여러 프로쎄스가 공유하면 이전 페지틀(old_page)의 내용을 새로 할당한 페지틀(new_page)로 복사한다. 경쟁조건을 방지하기 위해 복사작업을 시작하기 전에 old_page의 사용회수를 증가시킨다.

이전 페지가 령페지(zero_page)면 memset마크로를 리용하여 새로운 프레임을 0으로 채운다. 령페지가 아니면 memcpy마크로를 사용하여 페지틀의 내용을 복사한다. 령페지를 특별하게 취급할 필요는 없지만 주소를 덜 참조하게 만들어 극소형처리기의 하드웨어캐쉬를 보존하기때문에 체계성능을 향상한다.

페지를 할당하다가 프로쎄스를 차단할수도 있으므로 do_wp_page()함수는 함수가 시작한 이후에 페지표입구점이 변경되였는가를 검사한다.(pte와 *page_table의 값이 다를 때) 이 경우 새 페지틀을 해제하고 old_page의 사용회수를 감소시킨 후(앞에서 증가시킨것을 취소하려고) 완료한다. 모든 작업이 정상적이라면 최종적으로 새 페지틀의 물리주소를 페지표입구점에 기록하고 해당 TLB입구점을 비운다.

```
set_pte(pte, pte_mkwrite(pte_mkdirty(mk_pte (new_page,
   vma->vm_page_prot) ) ) ) ;
```

```
flush_tlb_page(vma, address) ;
lru_cache_add(new_page) ;
spin_unlock(&rnm->page_table_lock) ;
```

lru_cache_add()는 교체와 관련한 자료구조에 새 페지를을 삽입한다.

5) 불련속적인 기억기령역접근처리

불련속적인 기억기령역관리에서 핵심부는 불련속적인 기억기령역에 해당하는 페지표입구점을 갱신하는데서 아주 느리다고 설명하였다. 실제로 vmalloc()와 vfree()함수는 주핵심부페지표(즉 페지대역등록부인 init_mm.pgd와 이것의 자식페지표)만을 갱신한다. 그러나 핵심부초기화단계가 끝난 후에는 어떤 프로쎄스나 핵심부스레드도 주핵심부페지표를 직접 사용하지 않는다.

그러므로 핵심부방식에 있는 프로쎄스가 불련속적인 기억기령역을 처음 접근하는 경우에 대하여 보기로 하자. 선형주소를 물리주소로 변환하는 과정에서 CPU의 기억기관리유니트는 빈(null)페지표입구점을 만나 페지오유를 일으킨다. 례외가 핵심부방식에서 발생했고 오유가 발생한 선형주소가 TASK_SIZE보다 크므로 페지오유례외조종기는 이 특별한 경우를 구별할수 있다. 따라서 조종기는 해당 주핵심부페지표 입구점을 검사한다.

```
 vmalloo fault:
asm(''movl  %%cr3,0": ":=r" (pgd) ) ;
pgd = __pgd_offset (address) + (pgd_t *) __va(pgd) ;
pgd_k = init_mm.pgd + __pgd_offset (address) ;
if ( !pgd_present (*pgd_k) )
   goto no_context;
set_pgd (pgd, *pgd_k) ;
prnd = pmd_offset (pgd, address ) ;
prnd_k = pmd_offset (pgd_k, address ) ;
if ( !pmd_present (*pmd_k) )
   goto no_context ;
set_pmd (pmd, *pmd_k) ;
pte_k = pte_offset (pmd_k, address ) ;
if ( !pte_present ( *pte_k) )
   goto no_context ;
return ;
```

변수 pgd를 cr3등록기에 들어있는 현재프로쎄스의 페지대역등록부의 주소로 설정하고 변수 pgd_k를 주핵심부페지대역등록부로 설정한다. 페지오유가 발생한 선형주소에 해당하는 입구점이 비여있으면 no_content표식이 가리키는 코드로 이행한다.(《주소공

간밖의 잘못된 주소처리하기》 참고) 입구점이 비여있지 않으면 해당 입구점을 프로세스의 페지대역등록부의 해당 입구점으로 복사한다. 그 이후 작업전체를 주페지중간등록부 입구점과 주페지표입구점에서도 반복한다.

5. 프로세스주소공간의 생성과 제거

앞서 프로세스주소공간에서 설명한 프로세스에 새 기억기구역을 할당하는 6가지 전형적인 경우중 첫번째인 fork()체계호출을 하는 경우 자식프로세스용으로 새로 전체 주소공간을 생성해야 하는 경우와 반대로 프로세스가 완료될 때 핵심부는 프로세스의 주소공간을 파괴한다는 내용을 보았다. 여기서는 Linux에서 이 두 작업을 어떻게 수행하는가에 대하여 보기로 한다.

1) 프로세스주소공간의 생성

앞에서 본 《clone(), fork(), vfork() 체계호출》에서 언급한것처럼 핵심부는 새로운 프로세스를 생성하는동안 copy_mm()함수를 호출한다.(KERNEL\fork.c)

이 함수는 새로운 프로세스의 모든 페지표와 기억기서술자를 구성하여 프로세스주소공간을 생성한다.

일반적으로 매 프로세스는 자신만의 주소공간이 있지만 CLONE_VM기발을 설정하고서 clone()을 호출하여 가벼운 프로세스를 만들수도 있다. 가벼운 프로세스는 같은 주소공간을 공유한다. 즉 같은 페지집합에 접근할수 있다.

이전에 설명한 COW접근방법에 따르면 전통적인 프로세스는 부모의 주소공간을 상속받으며 페지는 읽기만 하는 동안은 공유된 상태로 남는다. 그러다가 이중 한 프로세스가 공유하는 페지중 하나에 쓰기를 하면 해당 페지를 복사한다. 어느 정도 시간이 지나면 새로 만든 프로세스는 부모프로세스의 주소공간과 다른 자신만의 주소공간을 보유하게 된다. 반면에 가벼운 프로세스는 부모프로세스의 주소공간을 그대로 사용한다.

Linux는 이것을 주소공간을 복사하지 않고 간단하게 실현한다. 가벼운 프로세스는 일반 프로세스보다 상당히 빨리 만들수 있으며 부모와 자식프로세스가 조심해서 접근을 조절하는 한 기억기를 공유하는것 역시 리득이다.

flag변수에 CLONE_VM기발을 설정하여 clone()체계호출을 해서 새로운 프로세스를 만들면 copy_mm()은 부모(current)의 주소공간을 복제프로세스(tsk)에 그대로 준다.

```
if (clone_flags & CLONE_VM) {
        atomic_inc(&oldmm->mm_users);
        mm = oldmm;
        spin_unlock_wait(&oldmm->page_table_lock);
        goto good_mm;
  }
```

```
good_mm:
   tsk->mm = mm;
   tsk->active_mm = mm;
   return 0;
```

CLONE_VM기발을 설정하지 않으면 copy_mm()은(프로쎄스가 주소를 사용하기 전까지는 주소공간에 아무런 기억기도 할당하지 않더라도) 새로 주소공간을 만들어야 한다. 이 함수는 새로운 기억기서술자를 할당하고 이 주소를 새 프로쎄스서술자 tsk의 mm마당에 저장한 후 여러 마당을 초기화한다.

```
tsk->mm = kmem_cache_alloc(mm_cachep, SLAB_KERNEL) ;
tsk->active_mm = tsk->mm
memcpy(tsk->mm, current->mm sizeof(*tsk->mrn) ) ;
atomic_set (&tsk->mm->mm_users, 1) ;
atomic_set (&tsk->mm->mm_count, 1) ;
init_rwsern(&tsk->mm->mmap_sem) ;
tsk->mm->page_table_lock = SPIN_LOCK_UNLOCKED;
tsk->mm->pgd = pgd_alloc(tsk->mm) ;
```

pgd_alloc()마크로가 새 프로쎄스용으로 페지대역등록부를 할당한다는 사실은 기억할것이다.

다음으로 dup_mmap()함수를 호출하여 부모프로쎄스의 기억기구역과 페지표를 복제한다.

```
retval = dup_mmap(mm, oldmm);
if (retval)
   goto free_pt;
```

dup_mmap()함수는 새로운 기억기서술자인 mm을 전체 기억기서술자의 목록에 삽입한다.(kernel/fork.c)

그후 mm->mmap이 가리키는 곳부터 시작하여 부모프로쎄스가 소유하는 구역의 목록을 검색한다. vm_area_struct기억기구역서술자를 만날 때마다 이것을 복사하여 자식프로쎄스소유의 구역목록에 복사본을 넣는다.

dup_mmap()는 새로운 기억기구역서술자를 삽입한 다음 copy_page_range()를 (mm/memory.c) 호출하여 필요하다면 기억기구역에 있는 페지그룹을 배치하는데 필요한 페지표를 만들고 새 페지표입구점을 초기화한다.

특히 쓰기가능한 개인페지(VM_SHARE기발은 설정되지 않고 VM_MAYWRITE기발이 설정되여있는)에 해당하는 모든 페지들을 COW기구를 통해서 처리할수 있도록 부모와 자식 모두에 읽기전용으로 표시한다. 완료하기 전에 bulld_mmap_rb()함수를 호출해 자식프로쎄스의 기억기구역의 빨간색-검은색나무를 만든다. 마지막으로

copy_mm()은 자식의 기억기서술자에서 방식에 의존하는 부분을 초기화하는 copy_segments()를 호출한다. 기본적으로 부모프로쎄스가 자신에 맞춘 LDT를 가진다면 이것 역시 복사하여 자식에 할당한다.

2) 프로쎄스주소공간의 제거

프로쎄스가 완료될 때 핵심부는 exit_mm()함수를 호출해서 프로쎄스가 소유하던 주소공간을 해제한다.(KERNEL\exit.c)

mm_release()함수는 tsk->vfork_done 완료(completion)를 기다리며 잠들어있는 모든 프로쎄스를 깨운다.(KERNEL\fork.c)

일반적으로 완료하는 프로쎄스를 vfork()체계호출을 사용하여 만든 경우에만 해당 대기렬이 비여있지 않다.(《clone(), fork(), vfork() 체계호출》을 참고) 또한 처리기를 지연TLB방식으로 설정한다.

3) 동적기억구역관리

매 Unix프로쎄스는 《동적기억구역(heap)》이라는 프로쎄스의 동적기억기요청을 처리하기 위한 특수한 기억기구역을 가진다. 기억기서술자의 start_brk와 brk마당이 각각 동적기억구역의 시작과 끝주소를 저장한다.

다음 C서고함수는 프로쎄스가 동적기억기를 요청하고 해제할 때 사용한다.

malloc(size)

동적기억기를 size바이트만큼 요청한다. 성공적으로 할당하면 시작하는 기억기위치의 선형주소를 반환한다.

calloc(n, size)

크기가 size인 요소 n개로 구성된 배렬을 요청한다. 성공적으로 할당하면 배렬의 요소를 0으로 초기화하고 첫번째요소의 선형주소를 반환한다.

free(addr)

시작주소가 addr인 malloc()나 calloc()로 할당한 기억기구역을 해제한다.

brk(addr)

직접 동적기억구역의 크기를 변경한다. addr변수는 current->mm->brk의 새로운 값을 지정하며 결과값은 기억기구역의 새로운 끝주소이다.(프로쎄스는 요청한 addr값과 결과값이 일치하는가를 검사해야 한다.)

sbrk(incr)

brk()와 비슷하지만 변수 incr로 지정한 바이트만큼 동적기억구역크기를 늘이거나 줄인다는 차이가 있다.

여기서 brk()함수는 목록에 있는 다른 함수와 달리 체계호출로 실현하는 유일한 함수이다.

나머지함수는 C서고에서 brk()와 mmap()을 사용하여 실현한다.

사용자방식에 있는 프로쎄스가 brk()체계호출을 하면 핵심부는 sys_brk(addr)함수를 실행한다.(MM\nommu.c)

brk()체계호출은 기억기구역에서 동작하므로 전체 페지를 할당하거나 전체 페지를 해제한다. 그러므로 이 함수는 addr의 값을 PAGE_SIZE의 배수가 되도록 정렬한 후 그 값이 기억기서술자의 brk마당값과 같은가를 비교한다.

```
newbrk = ( addr + 0xfff) & Oxfffff000;
oldbrk = (mm->brk + 0xfff ) & Oxfffff000;
if (oldbrk == newbrk) {
   mm->brk = addr ;
   goto out ;
}
```

프로쎄스가 동적기억구역의 크기를 줄여줄것을 요청하면 sys_brk()는 do_munmap()를 호출하여 이 작업을 수행한 후 완료한다.

```
if (addr <= mm->brk) {
   if ( !do_munmap (mm, newbrk, oldbrk-newbrk) )
          mm->brk = addr ;
   goto out ;
}
```

반대로 프로쎄스가 동적기억구역의 크기를 늘여줄것을 요청하면 sys_brk()는 먼저 프로쎄스가 이것을 할수 있는가를 검사한다. 프로쎄스가 제한된 크기이상의 기억기를 할당받으려 하면 추가로 기억기를 할당하지 않고 기존의 mm->brk값을 반환한다.

```
rlim = current->rlim[RLIMIT_DATA].rlim_cur;
if (rlim < RLIM_INFINITY && addr - mm->start_data > rlirn)
goto out;
```

다음으로 확장한 동적기억구역이 프로쎄스가 소유한 다른 기억기구역과 겹치는가를 검사하고 겹치면 아무일도 하지 않고 돌아간다.

```
if (find_vma_interection(mm, oldbrk, newbrk+PAGE_SIZE))
   goto out ;
```

동적기억구역을 확장하기 전에 마지막으로 수행하는 검사는 동적기억구역을 확장할만큼 여유가상기억기가 충분한가를 검사하는것이다.(앞에서 본 《선형주소구간 할당》을 참고)

```
if (!vm_enough_memory (( newbrk-oldbrk ) >> PAGE_SHIFT ) )
   goto out ;
```

모두 이상이 없으면 MAP_FIXED기발을 설정한 상태에서 do_brk()함수를 호출한

다. 이 함수가 oldbrk값을 반환하면 할당에 성공한것이며 이 경우 sys_brk()는 addr값을 반환한다. 그렇지 않으면 이전의 mm->brk값을 반환한다.

```
if (do_brk (oldbrk, newbrk-oldbrk) == oldbrk)
    mm->brk = addr;
    gotoout;
```

do_brk()함수는 사실 do_mmap()함수를 닉명기억기구역만을 다루도록 단순하게 만든것이다.

이 함수를 호출하는것은 다음과 같다.

```
do_mmap(NULL, oldbrk, newbrk_oldbrrk, PROT_READ | PROT-WRITE | PROT-EXEC, MAP_FIXED | MAP_PRIVATE, 0)
```

물론 do_brk()는 기억기구역이 디스크에 있는 파일을 배치하지 않는다고 가정하기때문에 기억기구역에 대한 몇가지 검사를 하지 않아도 되므로 do_mmap()보다 약간 빠르다.

제 3 절. 신호

신호는 최초에 등장한 Unix체계에서 프로쎄스사이에 호상작용할수 있게 하려고 도입하였다. 핵심부는 체계에서 일어난 어떤 사건을 프로쎄스에 알릴 때도 신호를 사용한다. 신호는 근 30여년동안 약간의 변화만 있었을뿐이다.

1. 신호의 역할

신호(signal)는 프로쎄스나 프로쎄스그룹에 보낼수 있는 아주 짧은 통보문이다. 실례로 《clone()과 fork(), vfork()체계호출》에서 언급한 SIGCHLD마크로를 들수 있다. Linux에서 17이라는 값으로 확장하는 이 마크로는 자식프로쎄스중 하나가 중단되거나 완료했을 때 부모프로쎄스에 보내는 신호의 식별자이다. 이미 《페지오유례외조종기》에서 언급한 SIGSEGV마크로는 11이라는 값으로 확장하며 프로쎄스가 잘못된 기억기를 참조했을 때 해당 프로쎄스로 보내는 신호기식별자이다.

신호는 다음과 같은 두가지 중요한 목적이 있다.

- 특정사건이 발생한 사실을 프로쎄스에 알린다.
- 프로쎄스가 자신의 코드에 들어있는 신호조종기함수를 실행하게 한다.

물론 프로쎄스는 종종 일부 사건에 반응하여 특정함수를 실행해야 하므로 두가지 목적은 서로 배타적이지 않다.

표 3-5는 80x86용 Linux 2.6에서 처리하는 처음 31개 신호에 대하여 보여주었다. (SIGCHLD나 SIGSTOP 같은 몇개의 신호번호는 하드웨어구조에 따라 다르다. 게다가

SIGSTKFLT같이 특정한 하드웨어구조에서만 정의하는 신호도 있다.) 기본동작의 의미는 다음 부분에서 설명한다.

표 3-5. Linux/i386내의 처음 31개 신호

№	신호이름	기본 동작	설　명	POSIX
1	SIGHUP	완료	말단이나 프로쎄스의 조종권을 끊음	예
2	SIGINT	완료	건반에서 오는 새치기	예
3	SIGQUIT	쏟기	건반에서 오는 완료	예
4	SIGILL	쏟기	잘못된 명령	예
5	SIGTRAP	쏟기	오유추적을 위한 중단점	아니
6	SIGABRT	쏟기	비정상적인 완료	예
7	SIGBUS	쏟기	모선오유	아니
8	SIGFPE	쏟기	부동소수점례외	예
9	SIGKILL	완료	강제적인 프로쎄스완료	예
10	SIGUSR1	완료	프로쎄스에서 사용가능	예
11	SIGSEGV	쏟기	잘못된 기억기참조	예
12	SIGUSR2	완료	프로쎄스에서 사용가능	예
13	SIGPIPE	완료	아무도 읽지 않는 관에 쓰기	예
14	SIGALRM	완료	실시간박자수	예
15	SIGTERM	완료	프로쎄스완료	예
16	SIGSTKFLT	완료	보조처리기탄창오유	아니
17	SIGCHLD	무시	자식프로쎄스가 멈추거나 완료	예
18	SIGCONT	계속	프로쎄스가 멈춰있다면 실행재개	예
19	SIGSTOP	중지	프로쎄스실행중지	예
20	SIGTSTP	중지	말단에 의해 프로쎄스중지	예
21	SIGTTIN	중지	배경프로쎄스의 입력요청	예
22	SIGTTOU	중지	배경프로쎄스의 출력요청	예
23	SIGURG	무시	소케트에 긴급상태	아니
24	SIGXCPU	쏟기	CPU시간제한초과	아니
25	SIGXFSZ	쏟기	파일크기제한초과	아니
26	SIGVTALRM	완료	가상시계(virtual timer)	아니

27	SIGPROF	완료	프로파일시계(profile timer)	아니
28	SIGWINCH	무시	창문크기조정	아니
29	SIGIO	완료	입출력이 가능해짐	아니
29	SIGPOLL	완료	SIGIO와 동일	아니
30	SIGPWR	완료	전원공급고장	아니
31	SIGSYS	완료	잘못된 체계호출	아니
32	SIGUNUSED	완료	사용하지 않음	아니

이 표에서 설명하는 정규신호(rgular signal)외에도 POSIX표준에서는 실시간신호(real-time signal)라는 새로운 신호를 도입하였다. Linux에서 이 신호의 범위는 32부터 63까지이다. 실시간신호는 항상 대기렬에 쌓이므로 신호를 여러번 보내더라도 모두 받을수 있다는 점에서 정규신호와 크게 다르다.

반면에 같은 종류의 정규신호는 대기렬에 들어가지 않는다. 정규신호를 련이어 여러번 보내더라도 이것을 수신하는 프로쎄스는 하나만 받는다. Linux핵심부는 실시간신호를 사용하지는 않지만 여러 체계호출을 통해 POSIX표준을 완전히 지원한다. 프로그람 작성자는 여러 체계호출을 리용하여 신호를 전송하고 프로쎄스가 수신한 신호에 어떻게 반응하겠는가를 결정할수 있다. 표 3-6은 이러한 체계호출에 대하여 보여준다. 이 함수의 동작에 관해서는 후에 《신호처리관련체계호출》에서 자세히 설명한다.

표 3-6. 가장 중요한 신호관련체계호출

체계호출	설 명
kill()	프로쎄스에 신호를 전송한다.
sigaction()	신호와 관련한 동작을 변경한다.
signal()	sigaction()과 비슷하다.
sigpending()	대기중인 신호가 있는가를 검사한다.
sigprocmask()	차단할 신호목록을 수정한다.
sigsuspend()	신호를 기다린다.
Rt_sigaction()	실시간신호와 관련한 동작을 변경한다.
Rt_sigpending()	대기중인 실시간신호가 있는가를 검사한다.
Rt_sigprocmask()	차단할 실시간신호목록을 수정한다.
Rt_sigqueueinfo()	프로쎄스에 실시간신호를 전송한다.
Rt_sigsuspend()	실시간신호를 기다린다.
Rt_sugtimedwait	rt_sigsuspend()와 비슷하다.

신호의 중요한 특징은 어떤 상태인지 예측할수 없는 프로쎄스에 언제든지 신호를 보낼수 있다는 점이다. 실행중이 아닌 프로쎄스에 신호를 보내면 핵심부는 해당 프로쎄스가 다시 실행을 재개할 때까지 신호를 저장해야 한다. 신호를 차단(block)하면 해당 신호의 차단을 해제할 때까지 신호전송을 미루어야 하는데 이것은 신호를 전달할수 있는 상태가 되기 전에 발생한 신호의 처리문제를 더욱 복잡하게 만든다.

따라서 핵심부는 신호전송을 다음의 두 단계로 구별한다.

- **신호발생**

  핵심부는 새로운 신호를 보낸것을 나타내려고 신호를 받는 프로쎄스의 서술자를 갱신한다.

- **신호분배**

  핵심부는 신호를 수신한 프로쎄스의 실행상태를 바꾸거나 특정한 신호조종기를 실행하거나 이 두가지를 모두 실행해서 수신한 프로쎄스가 신호에 반응하게 만든다.

발생한 매 신호는 많아야 한번만 분배할수 있다. 신호는 소모성자원이다. 즉 신호기를 분배하고 나면 해당 신호의 기존 존재를 나타내는 모든 프로쎄스서술자정보를 지운다.

발생하였지만 아직 분배하지 않은 신호를 《대기중인 신호(pending signal)》이라고 한다. 어떤 순간이든 한 프로쎄스에서 같은 종류의 신호는 단 하나만 대기할수 있다. 같은 프로쎄스에 같은 종류의 신호가 추가로 대기하더라도 대기렬에 쌓지 않고 무시한다. 실시간신호는 이와 달리 같은 종류의 신호가 동시에 여러개 대기할수 있다.

일반적으로 신호기가 대기중인 상태로 남아있는 시간은 예측할수 없다. 이와 관련해서 핵심부에서는 다음과 같은 요인을 고려해야 한다.

일반적으로 신호를 현재실행중인 프로쎄스(즉 current프로쎄스)에만 분배한다.

지정한 신호를 프로쎄스가 선택적으로 차단하고있을수도 있다.(뒤에서 보게 되는 《차단된 신호집합수정》을 참고) 이 경우 프로쎄스는 차단을 해제할 때까지 해당 신호를 받지 않는다.

일반적으로 프로쎄스가 신호조종기함수를 실행할 때 해당 신호를 《마스크(mask)》한다. 즉 조종기가 완료할 때까지 해당 신호를 자동으로 차단한다. 따라서 신호조종기는 처리중인 신호가 다시 발생하더라도 새치기당하지 않으므로 조종기함수를 재진입가능하게 작성할 필요가 없다.

신호개념은 직관적이지만 핵심부에서 실현은 조금 복잡하다. 핵심부는 다음과 같은 작업을 수행해야 한다.

- 각 프로쎄스가 어떤 신호를 차단하고있는가를 기억한다.
- 핵심부방식에서 사용자방식으로 절환할 때마다 어느 프로쎄스에든 도착한 신호가 있는가를 검사한다. 대략 10ms간격으로 발생하는 시간새치기마다 대부분 이 작업을 수행한다.
- 무시할수 있는 신호인가를 판단한다. 다음에 나오는 조건을 모두 만족할 때

신호를 무시할수 있다.

- 신호를 받을 프로쎄스를 다른 프로쎄스가 추적하지 않는다.(프로쎄스서술자의 ptrace마당에 있는 PT_PTRACED기발이 0이다.)
- 신호를 받을 프로쎄스가 해당 신호를 차단하지 않는다.
- 신호를 받을 프로쎄스가 해당 신호를 무시한다.(프로쎄스가 명백히 무시하겠다고 지정했거나 해당 신호의 기본동작이 무시되고 이 동작을 바꾸지 않았다.)

- 신호를 처리한다. 이 경우 프로쎄스실행중 언제라도 프로쎄스를 조종기함수로 절환하고 조종기함수를 마친 후 원래의 실행문맥으로 복귀할수 있어야 한다.

여기에 덧붙여 Linux는 BSD와 System V에서 각기 다른 의미로 사용하는 신호도 고려해야 한다. 게다가 POSIX규약에도 따라야 한다.

2. 신호를 받을 때 수행하는 동작

프로쎄스는 신호에 3가지 방식으로 반응할수 있다.

1. 명시적으로 신호를 무시한다.

2. 해당 신호와 관련한 기본동작을 수행한다.(표 3-5참고) 핵심부에서 미리 정의하고있는 기본동작은 신호종류에 따라 다르며 다음중 하나이다.

완료(terminate)

프로쎄스를 완료한다.

쏟기(dump)

프로쎄스를 완료하고 가능하면 실행문맥을 포함하는 core파일을 만든다. 파일은 오유추적용도로 사용할수 있다.

무시(ignore)

신호를 무시한다.

중지(stop)

프로쎄스를 멈춘다. 즉 프로쎄스를 TASK_STOPPED상태로 만든다.

계속(continue)

프로쎄스가 멈추고있다면(TASK_STOPPED) 프로쎄스를 TASK_RUNNING상태로 만든다.

3. 해당 신호조종기함수를 호출하여 신호를 포착한다.

신호를 차단하는것과 신호를 무시하는것이 다르다는데 류의하자. 신호를 차단하는 동안에는 신호를 분배하지 않고 차단을 해제한 후에만 분배한다. 무시한 신호는 항상 분배하지만 더는 동작을 취하지 않는다.

SIGKILL과 SIGSTOP신호는 무시하거나 잡거나 차단할수 없으며 항상 기본동작을 실행해야 한다. 따라서 사용자는 적절한 특권만 있다면 어떤 프로쎄스든 프로그람이 어떻게 방어하든지 상관없이 SIGKILL과 SIGSTOP신호를 사용하여 프로쎄스를 완료하거나 멈출수 있다.

3. 신호관련자료구조

핵심부는 체계에 있는 모든 프로쎄스에 대해 현재 어떤 신호가 대기중이거나 마스크되여있는지 또한 모든 신호를 처리하는 방법에 관한 정보를 유지해야 한다. 이를 위해 핵심부는 프로쎄스서술자를 통해 접근할수 있는 여러 자료구조를 사용한다. 그림 3-12는 가장 중요한 자료구조를 보여준다.

표 3-7은 프로쎄스서술자에 있는 신호처리관련마당을 렬거한다.

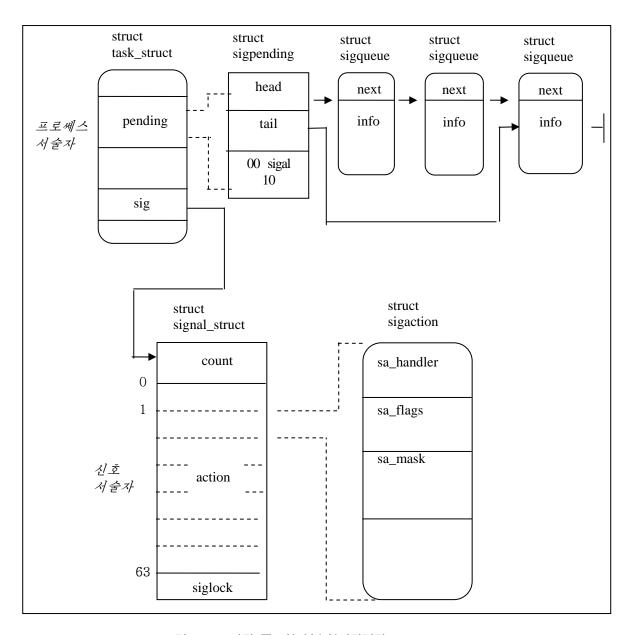



그림 3-12. 가장 중요한 신호처리관련자료구조

표 3-7. 신호처리관련 프로쎄스서술자마당

형	이 름	설명
Spinlock_t	sigmask_lock	pending과 blocked를 보호하는 스핀잠그기
struct signal_struct *	sig	프로쎄스의 신호서술자를 가리키는 지적자
sigset_t	blocked	차단된 신호의 마스크
struct sigpending	pending	대기중인 신호를 저장하는 자료구조
unsigned long	sas_ss_sp	교체신호조종기탄창의 주소
size_t	sas_ss_size	교체신호조종기탄창의 크기
int (*)(void *)	notifier	장치구동프로그람에서 프로쎄스의 일부 신호를 차단할 때 사용하는 함수를 가리키는 지적자
void *	notifier_data	바로 앞마당인 통보(notifier)함수에서 사용할수 있는 자료를 가리키는 지적자
sigset_t *	notifier_mask	통보문함수를 통해 구동프로그람에서 차단하는 신호의 비트마스크

blocked마당은 현재프로쎄스가 마스크한 신호를 저장한다. 이 마당의 형은 각 신호종류마다 한 비트씩 할당하는 비트의 배렬 sigset_t이다.

매 unsigned_long수자는 32bit이므로 Linux에서 선언할수 있는 신호의 최대 개수는 64이다.(_NSIG마크로가 이 값을 나타낸다.) 번호가 0인 신호는 없기때문에 sigset_t변수의 비트색인값에 1을 더하면 해당 신호번호가 된다. 1에서 31까지는 표 3-5에 렬거한 신호에 해당하고 32에서 64까지는 실시간신호에 해당한다.

프로쎄스서술자의 sig마당은 프로쎄스가 매 신호를 처리할 방법을 서술하는 신호서술자(signal descriptor)를 가리킨다. 이 서술자는 다음과 같이 정의하는 signal_struct구조체에 들어간다.

1절의 《clone(), fork(), vfork() 체계호출》에서 언급한것처럼 CLONE_SIGHAND 기발을 설정하여 clone()체계호출을 하면 이 자료구조를 여러 프로쎄스사이에서 공유할수 있다. count마당은 signal_struct자료구조를 공유하는 프로쎄스의 개수를 나타내고 siglock마당은 이 자료구조에 배타적인 접근을 하는데 사용한다. action마당은 매 신호의 처리방법을 지정하는 k_sigaction구조체 64개의 배렬이다.

어떤 방식에서는 신호에 핵심부에서만 볼수 있는 속성을 부여한다. 이런 신호속성은 k_sigaction구조체에 저장한다. 이 구조체는 사용자방식프로쎄스에서 볼수 없는 속성과 사용자방식프로쎄스가 볼수 있는 모든 속성을 담은 좀 더 포괄적인 sigaction구조체를 함께 포함한다. 80x86환경에서 사용자방식프로쎄스는 모든 신호속성을 볼수 있다. 따라서 k_sigaction구조체는 아래에 렬거한 마당을 담은 sigaction형의 sa구조체 하나만 포함한다.

1) sa_handler 또는 sa_sigaction

둘다 구조체내에 있는 동일한 마당을 가리킨다. 이 마당은 수행할 동작의 류형을 지정한다. 이 값은 신호조종기를 가리키는 지적자이거나 기본동작을 실행해야함을 의미하는 SIG_DFL(값이 0이다), 신호를 무시함을 의미하는 SIG_IGN(값이 1이다.)일수 있다. 이 마당의 서로 다른 두 이름은 신호조종기의 서로 다른 두 류형에 해당한다.

sa_flags

이 기발의 집합은 신호를 처리하는 방법을 나타낸다. 표 3-8에서 일부 기발에 대하여 볼수 있다.

sa_mask

이 sigset_t변수는 신호조종기를 실행할 때 마스크할 신호를 지정한다.

표 3-8. 신호처리방법을 지정하는 기발

기발이름	설명
SA_NOCLDSTOP	프로쎄스가 멈출 때 부모에 SIGCHLD를 보내지 않는다.
SA_NODEFER, SA_NOMASK	신호조종기를 실행할 때 신호를 마스크하지 않는다.
SA_RESETHAND, SA_ONESHOT	한번 신호조종기를 실행한 후에는 기본동작으로 돌아간다.
SA_ONSTACK	신호조종기용으로 교체탄창(alternate stack)을 사용한다.
SA_RESTART	새치기된 체계호출을 자동으로 재시작한다.
SA_SIGINFO	신호조종기에 추가정보를 제공한다.

현재 대기중인 신호를 관리하는데는 프로쎄스서술자에 있는 pending마당을 사용한다. 이 마당은 다음과 같이 정의하는 struct sigpending자료구조로 구성한다.

signal마당은 프로쎄스에 대기중인 신호를 나타내는 비트마스크이고 head와 taill은 《대기중인 신호대기렬(pending signal queue)》의 첫번째와 마지막항목을 가리키는 지적자이다. 이 대기렬은 struct sigqueue자료구조의 목록으로 실현한다.

nr_queued_signals변수는 대기렬에 있는 항목의 개수를 저장하고 max_queued_signals는 대기렬의 최대길이를 정의한다.(기본값은 1024이지만 체계관

리자가 /proc/sys/kernel/rtsig_max값을 쓰거나 적절히 sysctl()체계호출을 사용하여 바꿀수 있다.)

siginfo_t는 발생한 특정한 신호 하나에 관한 정보를 저장하는 128byte크기의 자료구조이다. 여기에는 다음과 같은 항목이 있다.

si_signo

신호번호

si_errno

신호를 발생시킨 명령의 오유코드, 오유가 발생하지 않았다면 0이다.

si_code

누가 신호를 발생시켰는가를 구별하는 코드(표 3-9참고)

표 3-9. 중요한 신호전송자코드

코드이름	전송자
SI_USER	kill()과 raise()
SI_KERNEL	일반핵심부함수
SI_TIMER	시계만료
SI_ASYNCIO	비동기적인 입출력완료

_sifields

신호종류에 따라 다른 정보를 저장하는 공용체(union) 례를 들어 SIGKILL신호가 발생하면 이에 해당하는 siginfo_t자료구조의 이 마당에 신호전송자의 PID와 UID를 기록한다. 반면에 SIGSEGV신호가 발생하면 접근하려다 신호가 발생한 기억기주소를 저장한다.

2) 신호자료구조에 대한 연산

핵심부는 신호를 다룰 때 여러가지 함수와 마크로를 사용한다. 다음 설명에서 set는 sigset_t변수에 대한 지적자이고 nsig는 신호번호, mask는 unsigned long형의 비트마스크(bit mask)이다.

sigemptyset(set)와 sigfillset(set)

sigset_t변수에 있는 모든 비트를 0이나 1로 설정한다.

sigaddset(set, nsig)와 sigdelset(set, nsig)

sigset_t 변수에서 신호 nsig에 해당하는 비트를 각각 1이나 0으로 설정한다.

sigaddsetmask(set, mask)와 sigdelsetmask(set, mask)

sigset_t변수에서 mask에 해당하는 모든 비트를 각각 1이나 0으로 설정한다. 이 함수는 1에서 32사이의 신호에서만 사용할수 있다.

sigismemmber(set, nsig)

sigset_t변수에서 신호 nsig에 해당하는 비트값을 돌려준다.

sigmask(nslg)

신호 nsig에 해당하는 비트색인값을 만든다. 즉 핵심부는 sigset_t형의 변수에서 특정신호에 해당하는 비트프레임을 설정하거나 지우거나 검사할 때 이 마크로를 사용하여 해당 비트를 추출할수 있다.

slgandsets(d, s1, s2)와 slgorsets(d, s1, s2), signandsets(d, s1, s2)

s1과 s2가 가리키는 sigset_t형의 변수사이에 각각 론리 AND나 론리 OR, 론리 NAND연산을 수행한 후 결과를 d가 가리키는 sigset_t변수에 저장한다.

sigtestsetmask(set, mask)

mask에서 1로 설정하는 비트중 하나라도 sigset_t변수에서 설정하고있으면 1, 그렇지 않으면 0을 반환한다. 번호가 1에서 32사이인 신호에서만 사용할수 있다.

siginitset(set, mask)

sigset_t변수에서 신호 1에서부터 32에 해당하는 아래비트를 mask에 있는 비트값으로 초기화하고 신호 33에서부터 63에 해당하는 비트를 모두 0으로 지운다.

siginitsetinv(set, mask)

setsigset_t 변수에서 신호 1에서부터 32에 해당하는 아래비트를 mask에 있는 비트값에 보수를 취한 값으로 초기화하고 신호 33에서부터 63에 해당하는 비트를 모두 1로 설정한다.

signal_pending(p)

프로쎄스서술자 *p가 가리키는 프로쎄스에 차단하지 않는 대기중인 신호가 있으면 1(참), 그렇지 않으면 0(거짓)을 반환한다. 이 함수는 프로쎄스서술자의 sigpending마당을 검사함으로써 간단하게 실현한다.

recalc_sigpending(t)

다음과 같이 프로쎄스서술자 *t가 가리키는 프로쎄스의 sig와 blocked마당을 검사하여 차단하지 않는 대기중인 신호가 있는가를 검사하고 이에 따라 sigpending마당을 0이나 1로 설정한다.

```
ready = t->pending.signal.sig[1] & ~t->blocked. sig[1] ;
ready |= t->pending.signal.sig[0] & ~t->blocked. sig[0] ;
  t->sigpending = (ready != 0);
void recalc_sigpending(void)
{
        recalc_sigpending_tsk(current);
}

fastcall void recalc_sigpending_tsk(struct task_struct *t)
```

```
{
   if (t->signal->group_stop_count > 0 ||
        PENDING(&t->pending, &t->blocked) ||
        PENDING(&t->signal->shared_pending, &t->blocked))
           set_tsk_thread_flag(t, TIF_SIGPENDING);
   else
           clear_tsk_thread_flag(t, TIF_SIGPENDING);
}
```

flush_signals(t)

프로쎄스서술자 *t가 가리키는 프로쎄스로 전달한 모든 신호를 삭제한다.

t->sigpending과 t->pending.signal마당을 모두 지우고 대기중인 신호대기렬을 비운다.

4. 신호발생

핵심부나 프로쎄스가 다른 프로쎄스로 신호를 보내면 핵심부는 send_sig_info()나 send_sig(), force_sig(), force_sig_info()함수를 호출하여 신호를 발생시킨다. 이 함수는 필요한대로 프로쎄스서술자를 갱신하여 앞서 《신호의 역할》에서 설명한 신호처리의 첫번째 단계를 완수한다. 이 함수는 신호를 분배하는 두번째 단계를 직접 수행하지는 않지만 신호의 종류와 프로쎄스의 상태에 따라 프로쎄스를 깨우거나 강제로 신호를 받게 할수 있다.

1) send_sig_info()와 send_sig()함수

sig

신호번호

info

siginfo_t표의 주소이거나 의미가 특별한 두가지 값(사용자방식프로쎄스가 신호를 보냈음을 의미하는 0과 핵심부방식이 보냈음을 의미하는 1)중 하나이다.

t

신호를 받을 프로쎄스의 프로쎄스서술자를 가리키는 지적자이다.

send_sig_info()함수는 먼저 변수가 정확한가를 검사한다.

```
if (sig < 0 || sig > _NSIG)
        return -EINVAL;
```

다음으로 사용자방식프로쎄스가 보낸 신호인가를 검사한다. 이것은 info값이 0이거나 siginfo_t표의 si_code마당이 0이나 부수일 때(정수값은 핵심부함수가 신호를 보냈다는것을 의미한다.)이다.

사용자방식프로쎄스에서 보낸 신호라면 이 작업을 할 권한이 있는가를 판단한다.

다음 조건가운데서 하나라도 해당할 때에만 신호를 분배한다.

· 신호를 보낸 프로쎄스의 소유자가 적합한 특질을 갖추어야 한다.(이것은 보통 체계관리자가 신호를 발생시켰다는것을 의미한다.)

· 신호가 SIGCONT이고 신호를 받는 프로쎄스와 보내는 프로쎄스가 같은 가입대화조종안에 있다.

· 두 프로쎄스가 같은 사용자소유이다.

sig변수값이 0이면 신호를 발생시키지 않고 즉시 함수를 완료한다. 0은 유효한 신호번호가 아니므로 신호를 보내는 프로쎄스가 받을 프로쎄스에 신호를 전달하는데 필요한 권한이 있는가를 검사할수 있게 한다. 또한 신호를 받는 프로쎄스가 TASK_ZOMBIE상태에 있어도 즉시 함수를 완료한다. 이것은 siginfo_t표의 해제여부를 검사하여 알수 있다.

```
if (!sig || !t->sig)
return 0;
```

이제야 핵심부가 기초적인 검사를 끝냈으므로 신호와 관련한 자료구조를 다루기 시작한다. 경쟁조건을 막으려고 새치기를 금지하고 신호를 받는 프로쎄스의 스핀잠그기 하나를 획득한다.

```
spin_lock_irqsave (&t->sigmask_lock, flags) ;
```

일부 종류의 신호는 신호를 받는 프로쎄스에서 대기중인 다른 신호를 없앨수도 있다. 따라서 다음경우 가운데서 하나에 해당하는가를 검사한다.

· sig가 SIGKILL이나 SIGCONT신호인 경우. 신호를 받는 프로쎄스가 멈춘상태이면 프로쎄스를 TASK_RUNNING상태로 바꿔서 이 프로쎄스가 do_exit()함수를 실행하거나 (SIGKILL)실행을 계속하게 한다.(SIGCONT)

나아가서 신호를 받는 프로쎄스에 SIGSTOP이나 SIGTSTP, SIGTTOU, SIGTTIN신호가 대기중이면 이것을 모두 제거한다.

```
if (t->state == TASK_STOPPED)
        wake_up_process(t);
t->exit_code = 0;
rm_sig_from_queue (SIGSTOP, t) ;
rm_sig_from_queue (SIGTSTP, t) ;
rm_sig_from_queue (SIGTTOU, t) ;
rm_sig_from_queue (SIGTTIN, t) ;
```

rm_sig_from_queue()함수는 t->pending.signal에서 첫번째 변수로 전달하는 신호번호에 해당하는 비트프레임 0으로 지우고 프로쎄스의 대기중인 신호대기렬에서 해당 신호번호에 해당하는 항목을 모두 제거한다.

· SIGSTOP이나 SIGTSTP, SIGTTIN, SIGTTOU중 하나인 경우. 신호를

받는 프로세스에 SIGCONT신호가 대기중이면 이것을 제거한다.

```
rm_sig_from_queue(SIGCONT, t);
```

다음으로 send_sig_from_info()함수는 새로운 신호를 즉시 처리할수 있는가를 검사한다. 이 경우 이 신호의 분배단계도 처리한다.

```
if (ignored_signal(sig, t)) {
    spin_unlock_irqrestore (&t->sigmask_lock, flags) ;
    return 0;
}
```

ignore_signal()함수는 《신호의 역할》에서 언급한 신호를 무시하는 3가지 조건을 모두 만족하면 1을 반환한다. 그러나 POSIX요구사항을 따르기 위해 SIGCHLD신호는 특별하게 처리한다. POSIX에서는 SIGCHLD신호를 무시하도록 직접 설정한것과 기본동작을 그대로 실행하도록 남겨둔것을 구별한다.

부모프로세스가 명시적으로 SIGCHLD신호를 무시하겠다고 설정하면 핵심부는 자식프로세스가 완료하면 이것을 깨끗이 정리하여 좀비가 되는것을 막는다. 그러므로 ignored_signal()함수는 다음과 같이 동작한다. 신호를 명시적으로 무시하면 0을 반환하지만 기본동작이 《무시》되면서 기본동작을 바꾸지 않았으면 1을 반환한다.

ignored_signal()함수가 1을 반환하면 신호를 받는 프로세스의 siginfo_t표를 갱신하면 안되고 send_sig_info()함수는 완료한다. 해당 신호는 더는 대기중이 아니므로 비록 받는 프로세스가 이 신호를 볼수 없지만 신호를 효과적으로 분배한것으로 된다.

ignored_signal()함수가 0을 반환하면 후에 신호의 분배단계를 처리하게 해야 한다. 따라서 send_sig_info()는 신호를 받는 프로세스가 자기에게 새로운 신호가 왔다는것을 알수 있도록 프로세스의 자료구조를 갱신해야 한다. send_sig_info()은 같은 신호가 이미 대기중이면 그냥 완료한다. 사실 발생한 정규신호는 실제로 대기렬에 계속 쌓이지 않으므로 프로세스의 대기중인 신호대기렬에서 모든 정규신호는 많아야 하나만 있을수 있다.

```
if (sig < 32 && sigismember (&t->pending.signal, sig)) {
spin_unlock_irqrestore(&t->sigmask_lock, flags);
    return 0;
}
```

이제 send_sig_info()함수는 신호를 받는 프로세스의 대기중인 신호대기렬에 새 항목을 추가해야 한다. 이것은 send_signal()함수를 호출해서 수행한다.

```
retval = send_signal(sig, info, &t->pending) ;
```

send_signal()함수는 대기중인 신호대기렬의 길이를 검사한 후 새로운 sigqueue 자료구조를 추가한다.

```
if (atomic_read(&t->user->sigpending) <
                t->rlim[RLIMIT_SIGPENDING].rlim_cur)
```

```
        q = kmem_cache_alloc(sigqueue_cachep, GFP_ATOMIC);
    if (q) {
        q->flags = 0;
        q->user = get_uid(t->user);
        atomic_inc(&q->user->sigpending);
        list_add_tail(&q->list, &signals->list);
        switch ((unsigned long) info) {
        case 0:
            q->info.si_signo = sig;
            q->info.si_errno = 0;
            q->info.si_code = SI_USER;
            q->info.si_pid = current->pid;
            q->info.si_uid = current->uid;
            break;
        case 1:
            q->info.si_signo = sig;
            q->info.si_errno = 0;
            q->info.si_code = SI_KERNEL;
            q->info.si_pid = 0;
            q->info.si_uid = 0;
            break;
        default:
            copy_siginfo(&q->info, info);
            break;
        }
```

send_signal( )함수는 대기렬에 추가한 새 항목의 siginfo_t표를 설정한다.

send_signal()함수에 전달한 info변수는 이전에 만든 siginfo_t표를 가리키거나 상수 0(사용자방식프로쎄스에서 보낸 신호인 경우)이나 1(핵심부함수에서 보낸 신호인 경우)을 저장한다.

이미 대기렬에 max_queued_signals개의 항목이 있거나 새로 sigqueue자료구조를 할당할 기억기가 부족해서 대기렬에 새 항목을 추가하는것이 불가능하면 발생한 신호를 대기렬에 넣을수 없다. 신호가 실시간신호이고 신호를 대기렬에 넣으라고 직접 요청하는 체계호출(rt_sigqeueinfo()같은)을 통해 신호를 전달한 경우라면 send_signal()함수는 오유코드를 반환하고 그렇지 않으면 t->pending.signal의 해당 비트를 설정한다.

```
    if (sig >= SIGRTMIN && info && (unsigned long)info != 1
```

```
            && info->si_code != SI_USER)
          /*
       * 대기렬 넘침.  신호가 rt이고 kill()보다 다른것을 리용하는
       * 사용자에 의해 보내졌다면 과제를 중지.
          */
                return -EAGAIN;
   if (((unsigned long)info > 1) && (info->si_code == SI_TIMER))

          ret = info->si_sys_private;

 out_set:
   sigaddset(&signals->signal, sig);
   return ret;
```

대기중인 신호대기렬에 해당 항목을 추가할 공간이 없더라도 프로쎄스가 신호를 받을수 있게 하는것이 중요하다. 례를 들어 너무 많은 기억기를 소비하는 프로쎄스가 있다고 하자. 핵심부는 사용가능한 기억기가 없더라도 kill()체계호출이 성공하도록 해야 한다. 그렇지 않으면 체계관리자가 해당 프로쎄스를 완료하지 못해 체계를 복구하지 못할수 있다.

send_signal()함수가 성공적으로 끝나고 해당 신호를 차단하고있지 않으면 신호를 받는 프로쎄스는 살펴야 하는 새로운 대기중인 신호를 가지게 된다.

```
  if (!retval && !sigismember(&t->blocked, sig) )
 signal_wake_up(t) ;
```

signal_wake_up()함수는 3가지 일을 수행한다.

1. 신호를 받는 프로쎄스의 sigpending기발을 설정한다.

2. 신호를 받는 프로쎄스가 이미 다른 CPU에서 실행중인가를 확인한다. 이 경우에는 그 CPU로 처리기간 새치기를 보내서 현재프로쎄스를 강제로 다시 순서짜기하게 한다.(《처리기간새치기처리》참고) 프로쎄스는 schedule()함수에서 돌아올 때 대기중인 신호가 있는가를 확인하므로 처리기간 새치기는 신호를 받는 프로쎄스가 이미 실행중인 경우 새로운 대기중인 신호를 빨리 인식하게 한다.

3. 신호를 받는 프로쎄스가 TASK_INTERRUPTIBLE상태인가를 검사한다. 이 경우에는 wake_up_process()함수를 호출하여 프로쎄스를 깨운다.

wake_up_process()함수는 우에서 서술한바가 있다.

마지막으로 send_sig_info()함수는 새치기를 다시 허용하고 스핀잠그기를 해제한 후 send_signal()의 오유코드를 반환한다.

spin_unlock_irqrestore(&t->sigmask_lock, flags);

return retval;

send_sig()함수는 send_sig_info()함수와 비슷하다. 이 함수는 info대신 priv기발을 변수로 받는다. 이 기발은 핵심부가 신호를 보내는 경우 1, 프로쎄스가 보내는 경우에는 0이다. 핵심부는 send_sig()함수를 send_sig_info()의 특별한 경우로 처리하여 실현한다.

return send_sig_info(sig, (void *) (long)(priv != 0) , t) ;

2) force_sig_info()와 force_sig()함수

force_sig_info(sig, info, t)함수는 핵심부가 프로쎄스에 무시하거나 차단할수 없는 신호를 보낼 때 사용하는 함수이다.(KERNEL\signal.c)

이 함수는 send_sig_info()와 동일한 변수를 받는다. force_sig_info()함수는 신호를 받는 프로쎄스서술자 t에 들어있는 sig마당이 가리키는 signal_struct자료구조를 리용해서 작업한다.

force_sig()는 force_sig_info()와 비슷하다. 이 함수는 핵심부가 신호를 보내는 용도로만 제한된다. 핵심부는 이 함수를 force_sig_info()함수의 특별한 경우로 처리하여 실현한다.

5. 신호분배

핵심부가 신호의 도착을 알아차리고 바로 앞에서 본 함수중 하나를 호출하여 신호를 받는 프로쎄스의 프로쎄스서술자에 필요한 작업을 하였다고 하자. 그러나 그 당시에 해당 프로쎄스가 CPU에서 실행중이 아니라면 핵심부는 신호를 분배하는 일을 뒤로 미룬다. 이제는 핵심부가 프로쎄스의 대기중인 신호를 처리하려고 수행하는 작업에 대하여 보기로 하자.

핵심부는 프로쎄스가 사용자방식에서 실행을 재개하도록 하기 전에 프로쎄스서술자의 sigpending기발의 값을 검사한다. 따라서 핵심부는 새치기나 례외처리를 마칠 때마다 대기중인 신호가 있는가를 검사한다.

차단하지 않는 대기중인 신호를 처리하기 위해 핵심부는 do_signal()함수를 호출한다.

이 함수는 다음과 같은 두개의 변수를 받는다.

regs

현재프로쎄스의 사용자방식등록기내용을 저장한 탄창령역의 주소

oldset

do_signal()함수가 차단된 신호의 비트마스크배렬을 저장할 변수의 주소. 이 비트마스크배렬을 저장할 필요가 없을 때에는 NULL 값을 담는다.

do_signal()함수는 먼저 자기가 새치기를 통해 호출됐는지 검사해서 이 경우 그냥

완료한다. 그렇지 않으면 프로쎄스가 사용자방식에서 실행중일 때 례외가 발생하여 자기가 검사하는데 이 경우 다음코드를 실행한다.

```
if ((regs->xcs & 3) != 3)
return 1;
```

그러나 《체계호출재실행》에서 보지만 이것은 신호가 체계호출을 새치기할수 없다는 사실을 의미하지는 않는다.

oldset변수가 NULL이면 이것을 current->blocked마당주소로 초기화한다.

```
if (!oldset)
oldset = &current->blocked;
```

do_signal()함수의 핵심은 차단하지 않는 대기중인 신호가 없을 때까지 dequeue_signal()함수를 반복해서 호출하는 순환이다.

dequeue_signal()의 결과값을 변수 signr에 저장한다. 이 값이 0이면 대기중인 모든 신호를 처리하여 do_signal()을 마쳐도 된다는것을 의미한다. 결과값이 0이 아닌 동안은 처리해야 할 대기중인 신호가 있다는것을 의미하므로 do_signal()은 현재 신호를 처리한 후 다시 dequeue_signal()을 호출한다.

dequeue_signal()은 항상 번호가 가장 낮은 대기중인 신호를 먼저 고려한다. 그리고 자료구조를 갱신하여 해당 신호가 더는 대기중이 아니라고 표시한 후 그 번호를 반환한다. 이 작업은 current->pending.singal의 해당 비트를 지우는 일과 current->sigpending의 값을 갱신하는 일도 포함한다. mask변수에서 1인 비트는 차단하는 신호를 나타낸다.

현재 대기중인 신호와 차단하는 신호(mask의 보수)를 AND연산한다. 그래서 남는 것이 있다면 신호를 프로쎄스에 분배해야 한다. ffz()함수는 변수로 전달한 값에서 0이 아닌 첫번째 비트의 색인값을 반환한다. 이 값을 통해서 분배할 가장 낮은 신호번호를 계산한다.

do_signal()함수에서 dequeue_signal()이 돌려준 번호를 가진 대기중인 신호를 어떻게 처리하는가를 보기로 하자. 먼저 다른 프로쎄스가 수신프로쎄스인 current를 감시하고있는가를 검사한다. 이 경우 do_signal()함수는 신호를 처리함을 감시하는 프로쎄스에서 알수 있게 notify_parent()와 schedule()함수를 호출한다.

다음으로 do_signal()은 처리할 신호의 k_sigaction자료구조의 주소를 변수 ka에 저장한다.

```
ka = &current->sig->action[signr-1] ;
```

여기에 있는 내용에 따라 신호의 무시, 기본동작을 수행하거나 신호조종기를 수행하는 3가지 동작중 하나를 실행한다.

1) 신호무시

분배한 신호를 명시적으로 무시한다면 do_signal()함수는 일반적으로 순환을 계속

실행하여 대기중인 다른 신호를 처리한다. 여기에서는 앞서 설명한 한가지 레외에 대하여 보기로 한다.

```
if (ka->sa.sa_handler == SIG_IGN) {
if (signr == SIGCHLD)
        while (sys_wait4 (-1, NULL, WNOHANG, NULL) > 0)
                /* 아무 일도 하지 않음 */;
continue;
}
```

분배한 신호가 SIGCHLD라면 wait4()체계호출의 봉사루틴인 sys_wait4()를 호출하여 프로쎄스가 자식프로쎄스에 대한 정보를 읽도록 만들어서 완료한 자식프로쎄스에 남아있는 기억기를 깨끗이 정리하게 한다.(KERNEL\exit.c)

2) 신호의 기본동작수행

ka->sa.sa_handler가 SIG_DFL이라면 do_signal()함수는 해당 신호의 기본동작을 수행해야 한다. 유일한 레외로 앞에서 본 《신호를 받을 때 수행하는 동작》에서 설명한것처럼 수신프로쎄스가 init이라면 신호를 무시한다.

```
if (current->pid == 1)
continue;
```

다른 프로쎄스라면 신호의 종류마다 기본동작이 다르므로 signr값을 바탕으로 switch문을 수행한다.

기본동작이 《무시》인 신호는 쉽게 처리한다.

```
case SIGCONT: case SIGCHLD: case SIGWINCH:
continue;
```

기본동작이 《중지》인 신호는 현재프로쎄스를 멈추게 할수 있다. 이를 위해 do_signal()은 current의 상태를 TASK_STOPPED로 바꾸고 schedule()함수를 호출한다.(《schedule()함수》를 참고) 또한 부모프로쎄스가 SIGCHLD신호에 대해 SA_NOCLDSTOP기발을 설정하지 않았으면 current의 부모프로쎄스에 SIGCHLD신호를 보낸다.

```
case SIGTSTP: case SIGTTIN: case SIGTTOU:
if (is_orphaned_pgrp (current->pgrp) )
        continue;
case SIGSTOP:
current->state = TASK_STOPPED;
current->exit_code = signr;
if (current->p_pptr->sig && !(SA_NOCLDSTOP &
        current->p_pptr->sig->action [SIGCHLD-1].sa.sa_flags))
```

```
        notify_parent (current, SIGCHLD) ;
    schedule( );
    continue; .
```

SIGSTOP와 다른 신호에는 미묘한 차이가 있다. SIGSTOP은 항상 프로쎄스를 멈추게 하지만 다른 신호는 프로쎄스가 고아프로쎄스그룹 (orphaned process group)'에 속하지 않는 경우에만 멈추게 한다. POSIX표준에서는 어떤 프로쎄스그룹에 있는 프로쎄스중 하나라도 같은 대화조종의 프로쎄스그룹에 속하는 프로쎄스를 부모로 하는 한 고아프로쎄스그룹이 아니라고 정의한다.

기본동작이 《쏟기》인 신호는 프로쎄스의 현재 작업등록부에 core파일을 생성할수 있다. 이 파일은 프로쎄스의 주소공간과 CPU등록기내용전체를 출력한것이다. do_signal()은 core파일을 생성한 후 프로쎄스를 소멸한다. 남은 신호 18개의 기본동작을 《완료》하고 그냥 프로쎄스를 소멸한다.

```
    exit_code = sig_nr;
    case SIGQUIT: case SIGILL: case SIGTNP:
    case SIGABRT: case SIGFPE: case SIGSEGV:
    case SIGBUS: case SIGSYS: case SIGXCPU: case SIGXFSZ:
    if (do_coredump (signr, regs))
        exit_code != 0x80;
    default:
        sigaddset (&current->pending.signal.signr) ;
        recalc_sigpending(current) ;
        current->flags != PF_SIGNALED;
        do_exit(exit_code);
```

do_exit()함수는 신호번호와 coredump를 수행한 경우에 설정하는 기발을 OR연산을 한 값을 입력변수로 받는다. 이 값은 프로쎄스완료코드를 설정하는데 사용한다. 이 함수는 현재프로쎄스를 완료하므로 호출한곳으로 돌아오지 않는다.

3) 신호포착

신호에 별도의 조종기가 있다면 do_signal()함수는 handle_signal()을 호출해서 해당 조종기를 실행하게 한다.

```
    handle_signal(signr, ka, &info, oldset, regs);
    return 1;
```

do_signal()이 신호를 하나만 처리한 후에 완료한다는 점에 주의를 돌려야 한다.

대기중인 다른 신호는 다음에 do_signal()을 호출할 때까지 신경쓰지 않는다. 이러한 접근법은 신호를 순서대로 처리함을 보장한다.

신호조종기를 실행하려면 사용자방식과 핵심부방식사이를 절환할 때 탄창을 조작해

야 하므로 이것은 어느정도 복잡한 작업이다. 이제부터 이때 정확하게 무슨 일을 해야 하는가에 대하여 설명하려고 한다.

신호조종기는 사용자방식프로쎄스에서 정의한 함수로 사용자방식코드토막에 들어있다.

handle_signal()함수는 핵심부방식에서 동작하지만 신호조종기는 사용자방식에서 동작한다. 따라서 현재프로쎄스가 정상적인 실행을 재개하기 전에 사용자방식에서 신호조종기를 먼저 실행해야 한다. 게다가 사용자방식에서 핵심부방식으로 절환할 때마다 핵심부방식의 탄창은 지워지므로 핵심부가 프로쎄스의 정상적인 실행을 재개하도록 할 당시에는 핵심부방식탄창은 새치기된 프로그람의 하드웨어문맥을 더는 저장하지 않는다.

여기에 추가하여 신호조종기가 체계호출을 실행할수도 있으므로 더욱 복잡해진다. 이 경우에는 봉사루틴을 실행한 후에 조종이 새치기된 프로그람코드대신 신호조종기로 돌아가야 한다.

Linux에서는 핵심부방식에 저장한 하드웨어문맥을 현재프로쎄스의 사용자방식탄창으로 복사하는 방법을 채택하여 해결한다. 또한 신호조종기를 완료할 때 sigreturn() 체계호출을 자동으로 호출하여 하드웨어문맥을 핵심부방식탄창으로 복사하고 사용자방식탄창의 원본을 복구하도록 사용자방식탄창을 변경한다.

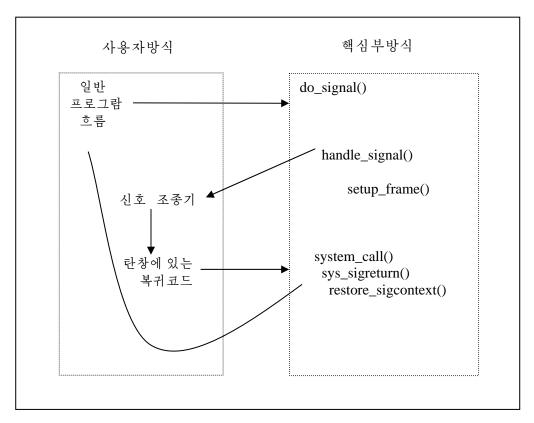



그림 3-13. 신호포착

그림 3-13은 신호를 잡는 작업에 관여하는 함수의 실행흐름을 보여준다. 먼저 프로쎄스로 차단하지 않는 신호를 전송한다. 새치기나 례외가 발생하면 프로쎄스는 핵심부방식으로 절환한다. 사용자방식으로 돌아가기 직전에 핵심부는 do_signal()함수를 실행하고 이 함수는(handle_signal()함수를 호출하여) 신호를 처리하고 (setup_frame()이나 setup_rt_frame()을 호출하여) 사용자방식탄창을 구성한다.

프로쎄스가 다시 사용자방식으로 절환할 때 프로그람계수기(program counter)를 조종기시작주소로 설정하므로 프로쎄스는 신호조종기를 시작한다. 이 함수가 완료하면 setup_frame()이나 setup_rt_frame()함수가 미리 사용자방식탄창에 설정해놓은 복귀코드를 실행한다. 이 복귀코드는 sigreturn()체계호출을 실행하고 이 체계호출의 봉사루틴은 정상 프로그람의 하드웨어문맥을 핵심부방식탄창으로 복사하고 사용자방식탄창을 원래상태로 복구한다.(restore_sigcontext()를 호출하여)

따라서 이 체계호출이 완료할 때 정상 프로그람에서 실행을 재개한다.

지금부터 이런 설계가 어떻게 동작하는가에 대하여 자세히 보기로 하자.

4) 프레임구성하기

handle_signal()함수는 프로쎄스 사용자방식탄창을 옳게 설정하려고 setup_frame()함수(siginfo_t표가 필요없는 신호일 때 뒤에 나오는 《신호처리관련 체계호출》을 참고)나 setup_rt_frame()함수(siginfo_t표가 필요한 신호일 때)를 호출한다. 핵심부는 두 함수중에서 어느것을 사용하겠는가를 결정하기 위해 그 신호에 해당하는 sigaction표의 sa_flags마당에 SA_SIGINFO기발이 설정되여있는가를 검사한다.

setup_frame()함수는 변수 4개를 받는다.

sig

신호번호

ka

신호와 관련한 k_sigaction표의 주소

oldset

차단하는 신호의 비트마스크배렬의 주소

regs

사용자방식등록기내용을 저장하는 핵심부방식탄창령역의 주소

setup_frame()함수는 사용자방식탄창에 프레임(frame)이라는 자료구조를 넣는다. 이 자료구조에는 신호를 처리하고 sys_sigreturn()함수로 정확하게 돌아올수 있게 하는데 필요한 정보가 들어간다. 프레임은 다음 마당을 포함하는 sigframe표이다.(그림 3-14 참고)

pretcode

신호조종기함수의 복귀주소: 같은 표의 retcode마당(목록 뒤부분에 나옴)을 가리킨다.

sig

신호번호: 이것은 신호조종기에 필요한 변수이다.

sc

핵심부방식으로 절환하기 직전의 사용자방식프로쎄스의 하드웨어문맥을 포함하는 sigcontext형구조체(current의 핵심부방식탄창에서 이 정보를 복사한다)이다. 여기에는 프로쎄스에서 차단하는 표준신호를 가리키는 비트배렬도 들어간다.

fpstate

사용자방식프로쎄스의 부동소수점등록기를 저장하는데 사용할수 있는 _fpstate형구조체.(《FPU와 MMX, XMM 등록기저장》을 참고)

extramask

차단하는 실시간신호를 지정하는 비트배렬.

retcode

sigreturn()체계호출을 하는 8byte코드: 신호조종기에서 복귀할 때 이 코드를 실행한다.

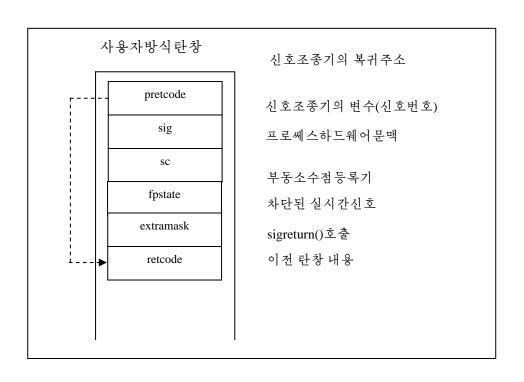



그림 3-14. 사용자방식탄창에 있는 프레임

setup_frame()함수는 먼저 get_sigframe()을 호출하여 프레임이 시작할 기억기위치를 계산한다.

이 기억기위치는 보통 사용자방식탄창에 있으므로 이 함수는 다음 값을 반환한다.

```
(regs->esp - sizeof(struct sigframe)) & 0xfffffff8
```

탄창은 낮은 주소쪽으로 커지므로 프레임시작주소는 현재 탄창의 맨우에 있는 주소에서 프레임크기를 뺀 결과를 8의 배수로 정렬해서 구한다.

다음으로 access_ok마크로를 사용하여 복귀주소를 검사한다. 이 주소가 유효하다면 __put_user()함수를 반복해서 호출하여 모든 프레임마당을 채운다. 이 작업이 끝나면 핵심부방식탄창의 regs령역을 수정하여 current가 사용자방식에서 실행을 재개하면 신호조종기로 조종이 넘어가게 만든다.

```
regs->esp = (unsigned long) frame;
regs->eip = (unsigned long) ka->sa.sa_handler;
```

setup_frame()함수는 핵심부방식탄창에 저장한 토막등록기를 기본값으로 재설정한 후 완료한다. 이제 신호조종기에 필요한 정보는 사용자방식탄창의 맨우에 있다.

setup_rt_frame()함수는 setup_frame()함수와 매우 류사하지만 사용자방식탄창에 신호와 관련된 siginfo_t표의 내용까지도 포함하는 확장프레임(extended frame, rt_sigframe자료구조에 저장한다.)을 넣는다.

5) 신호기발점검하기

handle_signal()함수는 사용자방식탄창을 구성한 후 신호와 관련된 기발값을 검사한다.

수신한 신호의 SA_ONESHOT기발이 1이라면 다음에 신호가 발생할 때 기본동작을 수행하도록 초기화해서 앞으로 같은 신호가 발생하더라도 신호조종기를 호출하지 않게 만든다.

```
if (ka->sa.sa_flags & SA_ONESHOT)
ka->sa.sa_handler = SIG_DFL;
```

나아가서 신호의 SA_NODEFER기발이 0이면 신호조종기를 실행하는 동안에 sigaction표의 sa_mask마당에서 지정하는 신호를 차단해야 한다.

```
if (!(ka->sa.sa_flags & SA_NODEFER)) {
spin_lock_irq(&current->sigmask_lock);
sigorsets (&current->blocked, &current->blocked, &ka->sa.sa_mask);
sigaddset (&current->blocked, sig) ;
recalc_sigpending (current) ;
spin_unlock_irq(&current->sigmask_lock) ;
}
```

앞에서 설명한것처럼 recalc_sigpending()함수는 프로쎄스에 차단하지 않는 대기중인 신호가 있는가를 검사하고 이에 따라 해당 sigpending마당을 설정한다.

handle_signal()은 완료하여 do_signal()함수로 되돌아가고 이 함수역시 완료한다.

6) 신호조종기시작

do_signal()함수가 완료하면 현재프로세스는 사용자방식에서 실행을 재개한다. 앞서 설명한 setup_frame()함수가 준비한대로 eip등록기는 신호조종기의 첫번째 명령을 가리키고 esp는 사용자방식탄창 맨우에 넣은 프레임의 첫번째 기억기위치를 가리킨다. 그 결과 신호조종기를 실행한다.

7) 신호조종기완료

신호조종기를 완료할 때 탄창의 맨우에 있는 복귀주소는 프레임의 retcode마당에 있는 코드를 가리킨다. siginfo_t표가 없는 신호인 경우 이 코드는 다음 기호언어명령과 같다.

```
popl %eax
movl $__NR_sigreturn, %eax
int $0x80
```

따라서 탄창에서 신호번호(즉 프레임의 sig마당)를 버리고 sigreturn()체계호출을 한다.

sys_sigreturn()함수는 사용자방식프로세스의 하드웨어문맥을 포함하는 pt_regs형 자료구조 regs의 주소를 계산한다.(arch\i386\kernel\signal.c)

이 함수는 esp마당에 들어있는 값을 통해 사용자방식탄창안에 있는 프레임주소를 계산하고 검사한다.

이제 신호조종기를 호출하기 전에 차단하던 신호의 비트배렬을 프레임의 sc마당에서 읽어들여 current의 blocked마당으로 복사한다. 결과적으로 신호조종기를 실행하려고 마스크한 모든 신호차단을 해제한다. 다음으로 recalc_sigpending()함수를 호출한다. (kernel\signal.c)

이 부분에서 sys_sigreturn()함수는 프로세스의 하드웨어문맥을 프레임의 sc마당에서 핵심부방식탄창으로 복사하고 사용자방식탄창에서 프레임을 제거해야 한다. 이 두가지 작업은 restore_sigcontext()함수를 호출하여 수행한다.

rt_sigqueueinfo()와 같은 체계호출을 사용하여 siginfo_t표가 필요한 신호를 보낸 경우라도 처리과정은 매우 비슷하다. 확장프레임내의 retcode마당에 들어있는 복귀코드는 rt_sigreturn()체계호출을 실행한다. 이것을 처리하는 sys_rt_sigreturn()봉사루틴은 프로세스의 하드웨어문맥을 확장프레임에서 핵심부방식탄창으로 복사하고 사용자방식탄창에서 확장프레임을 제거하여 원래 사용자방식탄창의 내용을 복구한다.

8) 체계호출재실행

종종 체계호출과 관련한 요청을 핵심부가 처리할수 없는 경우가 있다. 이런 일이 일어나면 체계호출을 실행한 프로세스는 TASK_INTERRUTIBLE이나 TASK_UNINTERRUPTIBLE상태가 된다.

프로세스가 TASK_INTERRUPTIBLE상태가 되고 다른 프로세스가 해당 프로세

스에 신호를 보내면 핵심부는 체계호출을 끝마치지 않은 상태에서 프로쎄스를 TASK_RUN상태로 만든다.(《새치기와 례외로부터 복귀》를 참고) 이런 일이 일어나면 체계호출봉사루틴은 작업을 끝마치지 않고 EINTR나 ERESTARTNOHAND, ERESTARTSYS, ERESTARTNOINTR오유코드를 반환한다. 프로쎄스는 사용자방식으로 돌아갈 신호를 전송한다.

실제로 이런 상태에서 사용자방식프로쎄스가 받을수 있는 유일한 오유코드는 EINTR이다. 이 오유코드는 체계호출을 끝까지 수행하지 않았다는것을 의미한다.(응용프로그람은 이 코드를 검사하여 체계호출을 다시 호출하겠는가를 결정할수 있다.) 나머지 오유코드는 핵심부가 신호조종기를 완료한 후에 자동으로 체계호출을 다시 호출하겠는가를 결정하기 위해 내부적으로 사용한다.

표 3-10은 끝마치지 못한 체계호출과 관련한 오유코드와 3가지 신호동작 각각에 대한 오유코드의 영향을 렬거한다. 여기에 나오는 용어의 의미는 다음과 같다.

- **완료**

체계호출을 자동으로 재실행하지 않는다. 프로쎄스는 사용자방식으로 돌아가 int $0x80다음에 나오는 명령부터 실행을 재개하고 eax등록기는 EINTR값을 저장한다.

- **재실행**

핵심부는 사용자방식프로쎄스가 체계호출번호를 eax등록기에 넣은 후 다시 int 0x80명령을 실행하도록 강제한다. 프로쎄스는 재실행사실을 모르며 프로쎄스에 오유코드를 전달하지도 않는다.

- **상태에 의존**

분배한 신호의 SA_RESTART기발이 1인 경우에만 체계호출을 재실행한다. 그렇지 않으면 체계호출은 EINTR오유코드를 반환하며 완료한다.

표 3-10. 체계호출의 재실행

신호 동작	오유코드와 체계호출실행에 미치는 영향			
	EINTR	ERESTARTSYS	ERESTARTHAND	ERESTARTNOINTR
기본	완료	재실행	재실행	재실행
무시	완료	재실행	재실행	재실행
포착	완료	상태에 의존	완료	재실행

신호를 분배할 때 핵심부는 체계호출을 재실행하기 전에 프로쎄스가 실제로 체계호출을 했는가를 확인해야 한다. 이곳이 바로 하드웨어문맥 regs와 orig_eax마당이 결정적인 역할을 하는 부분이다. 새치기조종기나 례외조종기가 시작할 때 이 마당을 어떻게 초기화하는가에 대하여 보기로 하자.

새치기

이 마당에는 새치기에 해당하는 IRQ번호에서 256을 뺀 값이 들어간다.

0x80례외

이 마당에는 체계호출번호가 들어간다.(《system_call()함수》를 참고)

다른 례외

이 마당에는 -1이 들어간다.(《례외조종기를 위한 등록기저장》을 참고) 따라서 orig_eax마당에 부수가 아닌 값이 들어있다면 체계호출처리중에 잠들어있던 TASK_INTERRUPTIBLE상태의 프로쎄스를 신호로 깨웠음을 의미한다. 이 봉사루틴은 체계호출이 새치기되였다는 사실을 알아차리고 이전에 언급한 오유코드중 하나를 반환한다.

만일 신호를 명시적으로 무시하거나 기본동작을 실행한다면 do_signal()함수는 오유코드를 분석하여 표 3-10에서 보여준바와 같이 끝나지 않은 체계호출을 자동으로 재실행하겠는가를 결정한다. 체계호출을 재실행해야 한다면 프로쎄스가 사용자방식으로 복귀할 때 eip는 int 0x80명령을 가리키고 eax는 체계호출번호를 담도록 하드웨어문맥 regs를 수정한다.

```
if (regs->orig_eax >= 0) {
    if (regs->eax == -ERESTARTNOHAND || regs->eax == -ERESTARTSYS || regs->eax == -ERESTARTNOINTR) {
        regs->eax = regs->orig_eax;
        regs->eip -= 2;
    }
}
```

regs->eax 마당을 체계호출봉사루틴의 복귀값으로 설정한다.(《system_call()함수》 참고)

조종기를 등록해서 신호를 잡은 경우라면 handle_signal()함수는 오유코드와 필요에 따라 sigaction표의 SA_RESTART기발을 분석하여 끝마치지 못한 체계호출의 재실행여부를 결정한다.

```
if (regs->orig_eax >= 0) {
switch (regs->eax) {
        case -ERESTARTNOHAND :
            regs->eax = -EINTR;
            break;
  case -ERESTARTSYS :
  if (!( ka->sa.sa_flags & SA_RESTART)) {
  regs->eax = -EINTR;
```

```
      break ;
      }
      /* 계속 진행 * /
      case -ERESTARTNOINTR :
      regs->eax = regs->orig_eax;
                  regs->eip -= 2 ;
        }
    }
```

체계호출을 재시작해야 한다면 handle_signal()은 do_signal()과 똑같이 진행하고 그렇지 않으면 사용자방식프로세스에 -EINTR오유코드를 반환한다.

6. 신호처리관련 체계호출

이 절의 앞부분에서 설명한바와 같이 사용자방식에서 실행하는 프로그람은 신호를 주고 받을수 있다. 따라서 이런 동작을 수행하기 위한 일련의 체계호출을 정의해야 한다. 같은 목적을 달성하는 여러 체계호출이 존재한다. 그 결과 체계호출의 일부는 전혀 사용하지 않기도 한다. 례를 들어 sys_sigaction()과 sys_rt_sigaction()은 거의 똑같다. 따라서 C서고에 들어있는 래퍼함수sigaction()은 sys_sigaction()대신에 sys_rt_sigaction()을 호출한다. 여기서는 몇가지 중요한 POSIX체계호출에 대하여 설명한다.

1) kill체계호출

kill(pid, sig)체계호출은 일반적으로 신호를 보낼 때 사용한다. 이것을 처리하는 봉사루틴은 sys_kill()함수이다.

정수형의 pid변수는 값에 따라 여러가지 의미를 나타낸다.

pid > 0

PID가 pid와 같은 프로세스에 sig신호를 보낸다.

pid = 0

이 체계호출을 한 프로세스와 같은 프로세스그룹에 있는 모든 프로세스에 sig신호를 보낸다.

pid = -1

교환기(PID 0)와 init(PID 1), current프로세스를 제외한 모든 프로세스에 신호를 보낸다.

pid < -1

pid프로세스그룹에 있는 모든 프로세스에 신호를 보낸다.

sys_kill()함수는 신호를 위한 최소한의 siginfo_t표를 만든 후 kill_something_info()함수를 호출한다.

```
info.si_signo = sig;
info.si_errno = 0;
info.si_code =  SI_USER;
info._sifields._kill._pid = current->pid;
info._sifields._kill._uid = current->uid;
return kill_something_info(sig, &info, pid) ;
```

kill_something_info()함수는 send_sig_info()함수를 호출하거나(프로쎄스하나에 보낸다.) kill_pg_info()함수를 호출한다.(모든 프로쎄스를 검색하여 신호를 받을 그룹에 있는 각 프로쎄스마다 send_sig_info()함수를 호출한다.)

kill()체계호출은 어느 신호이든 보낼수 있다. 심지어 실시간신호라고 부르는 32에서부터 63사이의 신호도 보낼수 있다. 그렇지만 앞서 《신호발생》에서 본것처럼 체계호출은 새로운 신호를 목적지프로쎄스의 대기중인 신호대기렬에 추가한다는 사실을 보장하지 않는다. 따라서 여러번 보낸 신호를 잃어버릴수도 있다. 실시간신호는 rt_sigqueueinfo()와 같은 체계호출을 사용해서 보내야 한다.(뒤에 나오는 《실시간신호를 위한 체계호출》을 참고)

System V와 BSD Unix변종에는 프로쎄스그룹에 직접 신호를 보내는 killpg()체계호출이 있다. Linux에서는 이 함수를 kill()체계호출을 리용하여 체계호출이 아닌 서고함수로 실현한다. 다른 변종으로 현재프로쎄스에(즉 이 함수를 호출하는 프로쎄스에) 신호를 보내는 raise()체계호출이 있다. Linux에서는 raise()도 서고함수로 실현한다.

2) 신호동작변경

sigaction(sig, act, oact)체계호출은 사용자가 신호에 대한 동작을 지정할수 있게 한다. 물론 신호동작을 정의하지 않으면 핵심부는 수신한 신호의 기본동작을 실행한다.

이것을 처리하는 sys_sigaction()봉사루틴은 변수 두개를 가지고 동작한다. (arch\i386\kernel\signal.c)

하나는 신호번호 sig이고 다른 하나는 새로운 동작을 정의하는 sigaction형의 표 act이다. 그리고 세번째 변수인 oact는 신호에 대한 이전 동작을 알아내는데 사용할수 있는 선택적인 출력변수이다.

이 함수는 먼저 act주소가 유효한가를 검사한다. 그리고 k_sigaction형의 변수 new_ka의 sa_handler와 sa_flags, sa_mask마당을 *act의 해당 마당값으로 채운다.

```
__get_user(new_ka.sa.sa_handler, &act->sa_handler);
__get_user(new_ka.sa.sa_flags, &act->sa_flags);
__get_user(mask, &act->sa_mask);
siginitset(&new_ka.sa.sa_mask, mask);
```

이 함수는 do_sigaction()을 호출해서 새로운 new_ka표를 current->sig->action의 sig-1번째 입구점으로 복사한다.(신호번호는 1부터 시작하므로 배렬에서의 위치보다

1만큼 크다.)

```
        k = &current->sig->action[sig-1];
        spin_lock(&current->sig->siglock);
    if (act) {
    *k = *act;
            sigdelsetmask(&k->sa.sa_mask, sigmask(SIGKILL) | sigmask(SI
        GSTOP));
            if (k->sa.sa_handler == SIG_IGN || (k->sa.sa_handler == SIG_
        DFL && (sig == SIGCONT || sig == SIGCHLD || sig == SIGWINC
        H)))
        {
          spin_lock_irq(&current->sigmask_lock);
          if (rm_sig_from_queue(sig, current))
          recalc_sigpending(current);
          spin_unlock_irq(&current->sigmask_lock);
    }
    }
```

POSIX표준에서는 기본동작이 《무시》인 신호의 동작을 SIG_IGN이나 SIG_DFL로 설정하면 종류가 같은 대기중인 신호를 무시하도록 한다. 나가서 해당 신호조종기에서 마스크할 신호로 무엇을 지정하든 SIGKILL과 SIGSTOP은 절대로 마스크할수 없다.

oact변수가 NULL이 아니라면 이전 sigactlon표의 내용을 변수가 가리키는 프로쎄스주소공간으로 복사한다.

```
    if (oact) {
    __put_user (old_ka.sa.sa_handler, &oact->sa_handler) ;
    __put_user (old_ka.sa.sa_flags, &oact->sa_flags) ;
    __put_user (old_ka.sa.sa_mask.sig[0], &oact->sa_mask) ;
    }
```

sigaction()체계호출을 사용하여 sigaction표의 sa_flags마당도 초기화할수 있다. 이장 앞부분에 있는 표 3-8에서 이 마당에서 사용할수 있는 값과 그 의미를 설명하였다.

초기 System V Unix의 변종은 signal()체계호출을 제공하였다. 이 체계호출은 아직까지 프로그람작성자들사이에서 널리 쓰인다. 최근 C서고는 sigaction()을 사용하여 signal()을 실현하지만 Linux는 여전히 오래된 C서고를 지원하여 sys_signal()봉사루틴을 제공한다.

```
    new_sa.sa.sa_handler = handler;
    new_sa.sa.sa_flags = SA_ONESHOT | SA_NOMASK;
```

```
ret = do_sigaction (sig, &new_sa, &old_sa) ;
return ret ? ret : (unsigned long) old_sa.sa.sa_handler;
```

3) 대기중인 차단된 신호검사

프로세스는 sigpending()체계호출을 사용하여 대기중인 차단된 신호 즉 차단된 동안 발생한 신호의 목록을 확인할수 있다. 해당 봉사루틴인 sys_sigpending()은 비트배렬을 복사할 사용자변수의 주소인 변수 set만 사용한다.

프로세스는 sigprocmask()체계호출을 사용하여 차단할 신호집합을 수정할수 있다. 이것은 표준신호(비실시간신호)에만 적용된다. 해당 봉사루틴인 sys_sigprocmask()는 변수 세개를 사용한다.(kernel\signal.c)

oset

프로쎄스주소공간내에 있는 이전 비트마스크를 저장할 비트배렬에 대한 지적자이다.

set

프로쎄스주소공간내에 있는 새로운 비트마스크를 담고있는 비트배렬에 대한 지적자이다.

how

다음 값중 하나를 담은 기발이다.

SIG_BLOCK

*set 비트마스크배렬은 차단할 신호의 비트마스크배렬에 추가할 신호를 지정한다.

SIG_UNBlOCK

*set 비트마스크배렬은 차단할 신호의 비트마스크배렬에서 제거할 신호를 지정한다.

SIG_SETMASK

*set 비트마스크배렬은 차단할 신호의 새로운 비트마스크배렬을 지정한다.

이 함수는 copy_from_user()를 호출하여 set변수가 가리키는 값을 변수 new_set에 복사하고 current의 차단할 표준신호의 비트마스크배렬을 변수 old_set로 복사한다. 다음으로 how기발에서 지정한대로 이 두변수를 통해서 작업한다.

```
if (copy_from_user (&new_set, set, sizeof (*set) ) )
return -EFAULT;
new_set &= ~(sigmask (SIGKILL) |sigmask (SIGSTOP) ) ;
spin_lock_irq (&current->sigmask_lock) ;
old_set = current->blocked.sig[0] ;
if (how == SIG_BLOCK)
sigaddsetrnask (&current->blocked, new_set);
else if (how == SIG_UNBLOCK)
```


교육성 교육정보쎈터

```
    sigdelsetmask (&current->blocked, new_set) ;
else if (how == SIG_SETMASK)
current->blocked.sig [0] = new_set;
else
return   EINVAL;
recalc_sigpending(current);
spin_unlock_irq( &current->sigmask_lock);
if (oset ) {
if ( copy_to_user ( oset, &old_set, sizeof (*oset)))
        return   EFAULT;
}
return 0;
```

4) **프로세스보류**

sigsuspend()체계호출은 mask변수가 가리키는 비트마스크배렬에서 지정한 표준신호를 차단한 후 프로세스를 TASK_INTERRUPTIBLE상태로 만든다. 프로세스는 무시하지 않고 차단하지 않는 신호를 수신한 경우에만 깨여난다.

해당 봉사루틴인 sys-sigsuspend()는 다음 코드를 실행한다.

```
mask &= ~(sigmask (SIGKILL) | sigmask(SIGSTOP));
spin_lock_irq(&current->sigmask_lock) ;
saveset = current->blocked;
siginitset (&current->blocked, mask) ;
recalc_sigpending (current) ;
spin_unlock_irq(&current->sigmask_lock) ;
regs->eax = -EINTR;
while (1) {
    current->state = TASK_INTERRUPTIBLE;
    schedule ( ) ;
    if (do_signal(regs, &saveset))
        return -EINTR;
}
```

schedule()함수는 실행할 프로세스를 선택한다. sigsuspend()체계호출을 실행한 프로세스가 후에 실행을 재개하면 sys_sigsuspend()는 프로세스를 깨운 신호를 분배하기 위해 do_signal()함수를 호출한다. 이 함수의 결과값이 1이면 해당 신호를 무시하지 않는다는 의미이므로 체계호출오유코드 EINTR를 반환하고 완료한다.

sigprocmask()와 sleep()을 결합하여 사용하면 걸보기에 같은 결과를 낼수 있기때

문에 sigsuspend()체계호출을 중복된것으로 볼수도 있지만 그렇지 않다. 프로쎄스를 번갈아 실행하기때문에 동작 A를 수행하는 체계호출을 실행한 후에 동작 B를 수행하는 체계호출을 실행하는것과 동작 A와 B를 동시에 수행하는 체계호출 하나를 실행하는것은 엄연히 다르다.

특별한 경우 sigprocmask()에서 차단을 해제한 신호를 sleep()함수를 호출하기 전에 분배할수도 있다. 이 경우 프로쎄스는 이미 분배한 신호를 기다리면서 영원히 TASK_INTERRUPTIBLE상태로 남을수도 있다. 반면에 sigsuspend()체계호출은 신호차단을 해제할 때부터 schedule()을 호출할 때까지 신호전송을 허용하지 않는데 이 시간동안은 다른 프로쎄스가 CPU를 사용할수 없기때문이다.

5) 실시간신호를 위한 체계호출

앞서 본 체계호출은 표준신호에만 해당하는것으로 사용자방식프로쎄스가 실시간신호를 다루려면 추가적인 체계호출이 필요하다.

실시간신호를 위한 체계호출중에서 rt_sigaction()과 rt_sigpending(), rt_sigprocmask(), rt_sigsuspend()는 앞서 설명한 내용과 비슷하므로 더 설명하지 않는다. 마찬가지로 실시간신호대기렬을 다루는 다른 두 체계호출도 자세히 설명하지 않는다.

rt_sigqueueinfo()

실시간신호를 보내서 신호를 받는 프로쎄스의 대기중인 신호대기렬에 추가한다.

rt_sigtimewait()

차단된 대기중인 신호하나를 분배하지 않고 대기렬에서 제거한 후 해당 신호번호를 반환한다. 차단된 대기중인 신호가 없으면 정해진 시간동안 현재프로쎄스를 보류한다.

제 4 절. 프로쎄스순서짜기

Linux는 다른 시분할체계와 마찬가지로 짧은 시간단위로 한 프로쎄스에서 다른 프로쎄스로 옮겨다니며 작업을 수행하여 걸보기에는 마치 여러 프로쎄스를 동시에 실행하는 효과를 낸다. 프로쎄스절환(process switch)자체는 이미 보았으므로 이 절에서는 언제 프로쎄스절환을 하고 어떤 프로쎄스를 선택할것인가에 관한 문제인 순서짜기(scheduling)에 대하여 본다.

1. 순서짜기방책

전통적인 Unix조작체계의 순서짜기알고리듬(scheduling algorithm)은 몇가지 목적을 달성해야 한다. 즉 프로쎄스반응시간은 짧아야 하며 배경작업의 처리량도 좋아야

한다. 그렇다고 CPU를 사용하지 못하는 프로세스가 있어서도 안되며 우선순위가 낮은 프로세스와 우선순위가 높은 프로세스를 융통성있게 처리하여야 한다. 순서짜기방책(scheduling policy)이란 새로 실행할 프로세스를 언제, 어떻게 선택하겠는가를 결정할 때 사용하는 일련의 규칙을 말한다.

Linux의 순서짜기는 시분할(time-sharing)기법을 토대로 한다. 즉 CPU시간을 얇은 쪼각(slice)으로 쪼개고 실행가능한 각 프로세스마다 쪼각을 하나씩 할당하여 프로세스 여러개를 《시간다중화(time multiplexlng)》방식으로 실행한다. 물론 한 처리기는 어느 순간이든 한 프로세스만 실행할수 있다. 현재 실행하고있는 프로세스가 완료하지 않은채 프로세스에 부여한 《시간쪼각 (tme slice)》 즉 《정량(quantum)》이 만료되면 프로세스절환이 일어날수 있다. 시분할은 시간새치기에 의존하므로 이 작업은 프로세스에 보이지 않는다. 프로그람은 CPU의 시분할을 위해 코드를 추가할 필요가 없다.

프로세스를 우선순위에 따라 순위를 매기는 작업 또한 순서짜기방책의 기반이 된다. 때로는 프로세스의 현재 우선순위를 알아내려고 복잡한 알고리듬을 사용하지만 최종 결과물은 같다. 이것은 각 프로세스가 CPU를 할당받는데 얼마나 적합한가를 나타내는 값이다.

Linux에서 프로세스의 우선순위를 동적으로 매긴다. 순서짜기프로그람(scheduler)은 프로세스가 무엇을 하는가를 계속 지켜보면서 주기적으로 프로세스의 우선순위를 조정한다. 이런 순서짜기프로그람은 오랜 시간 CPU를 사용하지 못한 프로세스의 경우 우선순위를 동적으로 높여서 밀어준다. 마찬가지로 오래동안 실행한 프로세스는 동적으로 우선순위를 낮춘다.

전통적으로 순서짜기에 관해 이야기할 때 프로세스를 입출력위주(I/O-bound)와 CPU위주(CPU-bound)로 분류한다. 전자는 입출력장치를 많이 리용하며 많은 시간을 입출력작업이 끝나기를 기다리는데 사용한다. 후자는 수자계산을 하는 프로그람처럼 CPU시간이 많이 필요한 응용프로그람이다. 다른 분류법으로 프로세스를 다음 3가지 종류로 구분하기도 한다.

- **호상작용프로세스**

  이 프로세스들은 끊임없이 사용자와 호상작용한다. 따라서 많은 시간을 건반이 눌리거나 마우스조작이 일어나길 기다리는데 소비한다. 프로세스는 입력을 받으면 빨리 깨여나야 한다. 그렇지 않으면 사용자는 체계의 반응속도가 늦다고 생각할것이다. 일반적으로 평균지연시간은 50~150ms사이여야 한다. 지연시간의 분산역시 이 범위안에 들어가야 한다. 그렇지 않으면 사용자는 체계가 정상이 아니라고 생각할것이다. 호상작용을 하는 전형적인 프로그람으로는 명령쉘(command shell)과 문서편집기, 도형프로그람이 있다.

- **일괄작업프로세스**

  이 프로세스들은 사용자와 호상작용을 요구하지 않는다. 그래서 때로는 배경작

업으로 실행하기도 한다. 이런 프로세스는 반응이 빠르지 않아도 된다. 전형적인 일괄작업 프로그람으로는 프로그람작성언어 콤파일러와 자료기지 검색엔진, 과학계산프로그람이 있다.

실시간프로세스

이 프로세스들은 순서짜기과 관련하여 많은 요구사항이 있다. 우선순위가 더 낮은 프로세스가 이런 프로세스를 차단해서는 안되며 짧은 반응시간을 보장하면서 이 시간편차를 최소화해야 한다. 전형적인 실시간프로그람으로는 영상과 음성관련응용프로그람, 로보트조종기 등이 있다.

여기서 설명한 두가지 분류법은 무관계하다. 례를 들어 일괄작업프로세스는 입출력위주일수도 있고(례를 들면 자료기지봉사기), CPU위주일수도 있다.(례를 들면 화상묘사프로그람) Linux는 순서짜기알고리듬에서 실시간프로그람을 명확하게 구별하지만 호상작용프로그람과 일괄작업프로그람을 구별할수 있는 방법은 없다. Linux는 호상작용하는 응용프로그람의 반응시간을 향상하기 위해 암시적으로 CPU위주프로세스보다는 입출력위주프로세스를 먼저 취한다.

프로그람작성자는 표 3-11에 렬거한 체계호출을 리용하여 순서짜기우선순위를 바꿀수 있다. 더 자세한 사항은 《순서짜기관련체계호출》에서 설명한다.

표 3-11. 순서짜기관련체계호출

체계호출	설 명
nice()	일반프로세스의 우선순위를 변경한다.
getpriority()	일반프로세스그룹의 최대우선순위값을 알아낸다.
setpriority()	일반프로세스그룹의 우선순위를 설정한다.
sched_getscheduler()	프로세스순서짜기방책을 알아낸다.
sched_setscheduler()	프로세스순서짜기방책과 우선순위를 설정한다.
sched_getparam()	프로세스순서짜기우선순위를 알아낸다.
sched_setparam()	프로세스순서짜기 우선순위를 설정한다.
sched_yield()	차단하지 않으면서 자진하여 처리기를 반납한다.
sched_get_priority_min()	지정한 순서짜기방책에 대한 최소우선순위값을 알아낸다.
sched_get_priority_max()	지정한 순서짜기방책에 대한 최대우선순위값을 알아낸다.

sched_rr_get_interval()	Round Robin방책의 시간정량값을 알아낸다.

표에 렬거한 체계호출중 대부분은 실시간프로쎄스(real-time process)에도 해당하므로 사용자는 이것을 리용하여 실시간응용프로그람을 개발할수 있다. 그러나 Linux핵심부는 비선취형(nonpreemptive)이므로 요구사항이 많은 대부분의 실시간응용프로그람을 지원하지 않는다.(《순서짜기알고리듬성능》 참고)

1) 프로쎄스선취

1장에서 언급한것처럼 Linux프로쎄스는 선취형(preemptive)이다. 어떤 프로쎄스가 TASK_RUNNING상태가 되면 핵심부는 이 프로쎄스의 동적우선순위가 현재 실행중인 프로쎄스의 우선순위보다 높은가를 검사한다. 더 높다면 current의 실행을 중단하고 순서짜기프로그람을 호출하여 다른 프로쎄스를 선택해서 실행한다.(보통은 방금 실행가능하게 된 프로쎄스를 선택한다.) 물론 프로쎄스가 자기에게 주어진 시간정량(time quantum)을 다 사용했을 때도 선취할수 있다. 이런 일이 벌어지면 현재프로쎄스의 need_resched마당을 1로 설정하여 시간새치기조종기가 끝날 때 순서짜기프로그람을 호출한다.

례를 들어 두 프로그람(문서편집기와 콤파일러)만 실행하고있는 상태를 생각해보자. 문서편집기는 호상작용하는 프로그람이므로 콤파일러보다 높은 동적우선순위를 가진다. 그렇지만 사용자는 생각하면서 멈추고 자료입력을 번갈아가며 수행하며 게다가 두 건을 입력하는 사이의 평균지연시간은 상대적으로 길기때문에 이 프로그람은 자주 보류된다. 그렇지만 사용자가 건을 입력하자마자 새치기가 발생하고 핵심부는 문서편집기프로쎄스를 깨운다.

또한 핵심부는 편집기의 동적우선순위가 현재 실행중인 프로쎄스(콤파일러)인 current의 우선순위보다 높다고 판단하여 현재프로쎄스의 need_resched 마당을 1로 설정한다. 따라서 핵심부가 새치기조종기를 마칠 때 순서짜기프로그람을 호출하게 만든다. 순서짜기프로그람은 편집기를 선택하고 프로쎄스절환을 한다. 그 결과 편집기는 아주 빠르게 실행을 재개하고 사용자가 입력한 글자를 화면에 보여준다. 입력한 글자를 처리한 후 문서편집기프로쎄스는 또 다른 건입력을 기다리며 보류상태가 되고 콤파일러프로쎄스는 실행을 재개할수 있다.

여기서 선취당한 프로쎄스는 보류되는것이 아니라는 점에 주의를 돌려야 한다. 이 프로쎄스는 여전히 TASK_RUNNING상태로 남아있으며 그저 CPU를 사용하지 않을뿐이다.

일부 실시간조작체계는 선취형핵심부(preemptive kernel)특징이 있다. 이것은 핵심부방식에서 실행중인 프로쎄스라도 사용자방식에서와 마찬가지로 아무 명령에서나 CPU를 가로챌수 있다는것을 의미한다. Linux핵심부는 선취형이 아니므로 사용자방식에 있을 때에만 프로쎄스를 선취할수 있다. 비선취형핵심부(nonpreemptive kernel)는

핵심부자료구조와 관련한 대부분의 동기화(synchronization)문제를 해결할수 있어 핵심부설계가 훨씬 간단하다.

2) 정량은 얼마나 길어야 하는가

정량(quantum)기간은 체계성능에 중요한 영향을 미치는데 너무 길어서도 안되고 너무 짧아서도 안된다.

정량기간이 너무 짧으면 프로쎄스절환때문에 발생하는 관리소비(overhead)가 지나치게 커진다. 례를 들어 프로쎄스절환을 하는데 10ms가 걸리고 정량역시 10ms로 설정한다고 하자. 그러면 프로쎄스절환을 하는데만 CPU시간의 50%를 쓰게 된다.

정량기간이 너무 길면 더는 프로쎄스를 동시에 실행하는것처럼 보이지 않는다. 례를 들어 정량을 5s로 설정한다면 실행가능한 각 프로쎄스는 5s동안 작업을 계속할수는 있겠지만 매우 오랜 시간동안 멈춰있어야 한다.(일반적으로 5s에 실행가능한 프로쎄스개수를 곱한 정도일것이다.)

정량기간이 길어지면 호상작용프로그람의 반응성이 떨어지는것으로 아는 사람도 있지만 일반적으로 틀린 말이다. 앞서 《프로쎄스선취》에서 설명한바와 같이 호상작용하는 프로쎄스는 상대적으로 우선순위가 높으므로 일괄작업프로쎄스가 아무리 긴 정량기간을 가진다고 하더라도 이것을 빠르게 선취한다.

정량기간이 너무 길면 체계의 반응성이 떨어지는 경우도 있다. 례를 들어 사용자 두명이 자신의 쉘프롬프트에서 동시에 명령을 내린다고 하자. 여기서 한 명령은 CPU위주프로그람이고 다른 명령은 호상작용프로그람이라고 하자. 두 쉘은 새로운 프로쎄스를 생성하여 사용자가 내린 명령의 실행을 해당 프로쎄스에 위임한다.

나아가서 이렇게 새로 만든 프로쎄스의 초기우선순위는 같다고 가정하자.(Linux는 실행한 프로그람이 일괄작업인지 호상작용을 하는지 미리 알지 못한다.) 순서짜기프로그람이 CPU위주프로쎄스를 선택한다면 이 프로쎄스가 시간정량을 모두 사용한 후에야 비로소 다른 프로쎄스를 실행할수 있다. 따라서 이 경우 정량기간이 길면 명령을 내린 사용자에게는 체계의 반응성이 떨어지는것처럼 보인다.

정량기간은 항상 절충안으로 선택한다. Linux가 채택한 제1규칙은 체계의 반응성을 좋게 유지하면서 정량기간을 가능한 길게 하는것이다.

2. 순서짜기알고리듬

Linux의 순서짜기알고리듬(scheduling algorithm)은 CPU시간을 《시기(epoch)》단위로 나누어 동작한다. 한 시기동안 모든 프로쎄스는 정해진 시간정량을 소유하며 이 기간을 그 시기가 시작할 때 계산한다. 일반적으로 서로 다른 프로쎄스는 시간정량기간이 서로 다르다. 시간정량값은 그 시기동안 프로쎄스에 할당하는 최대 CPU시간이다. 프로쎄스가 시간정량을 모두 소비하면 해당 프로쎄스를 선취하여 다른 실행할수 있는 프로쎄스로 교체한다. 물론 프로쎄스가 정량을 모두 소비하지 않는 한 순서짜기

프로그람은 한 시기안에 그 프로쎄스를 여러번 선택하여 실행할수도 있다. 례를 들어 프로쎄스가 입출력을 완료하길 기다리며 수행을 보류하면 자기한데 남은 시간정량을 보존하고있다가 같은 시기에 다시 선택될수 있다. 한 시기는 실행가능한 모든 프로쎄스가 자신의 정량을 완전히 소비할 때 끝난다. 이 경우 순서짜기프로그람 알고리듬은 모든 프로쎄스의 시간정량을 다시 계산하고 새로운 시기를 시작한다.

각 프로쎄스에는 기본시간정량(base time quantum)이 있다. 이것은 프로쎄스가 이전 시기에서 자신의 정량을 모두 소비하면 순서짜기프로그람이 프로쎄스에 할당하는 시간정량값이다. 사용자는 nice()와 setpriority()체계호출을 리용하여 자신이 실행하는 프로쎄스의 정량을 바꿀수 있다.(《순서짜기관련체계호출》을 참고) 새로 생성되는 프로쎄스는 항상 부모의 기본시간정량을 상속받는다.

INIT_TASK마크로는 프로쎄스0(교환기)의 기본시간정량값을 DEF_PRIOR로 설정한다. 이 마크로는 다음과 같이 정의한다.

```
#define DEF_COUNTER (10* HZ / 100)
```

IBM호환 PC에서는 HZ(시간새치기의 주파수를 나타내는 값)를 100으로 설정하기때문에(《프로그람가능한 간격시간》을 참고) DEF_PRIORITY는 10tick, 즉 대략 105미리초이다. Linux순서짜기프로그람은 실행할 프로쎄스를 선택할 때 프로쎄스의 우선순위를 고려해야 한다. 실제로 Linux에는 두가지 우선순위가 있다.

- **정적우선순위(static priority)**

  사용자가 실시간프로쎄스에 부여한것으로 값범위는 1부터 99까지이다. 순서짜기프로그람은 이 값을 절대로 바꾸지 않는다.

- **동적우선순위(dynamic priority)**

  일반 프로쎄스에만 해당되는것으로 기본적으로 기본시간정량(프로쎄스의 기본우선순위(base priority)라고도 부른다.)과 현재 시기에서 정량이 만료되기까지 프로쎄스에 남은 CPU시간의 tick수를 합한 값이다.

실시간 프로쎄스의 정적우선순위는 일반 프로쎄스의 동적우선순위보다 항상 높다. 순서짜기프로그람은 TASK_RUNNING상태에 있는 실시간프로쎄스가 없을 때에만 일반프로쎄스를 실행한다.

체계에는 언제나 실행가능한 프로쎄스가 적어도 하나는 있다. 이것은 바로 PID 0인 교환기(swapper)핵심부스레드로서 실행할 다른 프로쎄스가 없을 때에만 실행한다. 다중처리기체계의 모든 CPU에는 PID가 똑같이 0인 자신만의 핵심부스레드가 있다.

1) 순서짜기프로그람이 사용하는 자료구조

프로쎄스서술자에서 프로쎄스목록은 모든 프로쎄스서술자를 련결하며 실행대기렬(runqueue)목록은 모든 실행가능한 프로쎄스(즉 TASK_RUNNING상태에 있는 프로쎄스)의 프로쎄스서술자를 서로 련결한다고 설명하였다. 두 경우 모두 init_task프로쎄

스서술자가 목록의 머리역할을 한다.

프로세스서술자

각 프로세스서술자는 순서짜기와 관련한 여러 마당을 포함한다.

need_resched

ret_from_sys_call()이 schedule()함수를 호출해야 하는가를 나타내는 기발.

policy

순서짜기방식. 가능한 값은 다음과 같다.

SCHED_FIFO

먼저 들어온것이 먼저 나가는(FIFO, First In First Out)방식의 실시간프로세스. 순서짜기프로그람은 프로세스에 CPU를 할당할 때 실행대기렬 목록에서 해당 프로세스서술자를 현재 있는 위치에 그대로 둔다. 우선순위가 더 높은 실행가능한 실시간프로세스가 없다면 우선순위가 동일한 실행가능한 다른 프로세스가 있더라도 자신이 원하는동안 계속해서 CPU를 사용한다.

SCHED_RR

Round Robin방식의 실시간프로세스. 순서짜기프로그람은 프로세스에 CPU를 할당할 때 해당 프로세스서술자를 실행대기렬목록의 맨끝에 넣는다. 이 방책은 똑같은 우선순위가 동일한 SCHED_RR시간프로세스사이에서 공정한 CPU시간할당을 보장한다.

SCHED_OTHER

일반 시분할프로세스

policy마당에는 1bit크기인 SCHED_YIELD기발도 있다. 프로세스가 sched_yield()체계호출(입출력연산을 시작하거나 잠들지 않고서도 자진해서 처리기를 반납할수 있는 방법이다. 《순서짜기관련 체계호출》을 참고)을 하면 이 기발을 1로 설정한다. 또한 핵심부는 급하지 않은 작업을 오래동안 실행하면서 다른 프로세스에 실행할 기회를 주기 바랄 때마다 SCHED_YIELD기발을 설정하고 schedule()함수를 호출한다.

rt_priority

실시간프로세스의 정적우선순위. 유효한 우선순위의 범위는 1부터 99까지이다. 일반프로세스의 정적우선순위는 0으로 설정해야 한다.

counter

프로세스의 정량이 만료되기까지 프로세스에 남은 CPU시간의 tick수. 새로운 시기를 시작할 때 이 마당을 프로세스의 시간정량기간으로 설정한다. update_process_times()함수는 tick이 발생할 때마다 현재프로세스의 counter마당을 1씩 줄인다.

nice

새 시기를 시작할 때 프로쎄스가 가지는 시간정량의 길이를 지정한다. 이 마당은 -20에서 19사이의 값을 가지며 부수값은 《높은 우선순위》프로쎄스에, 정수값은 《낮은 우선순위》프로쎄스에 해당한다. 기본값은 보통 프로쎄스에 해당하는 0이다.

CPUs_allowd

프로쎄스를 실행할수 있도록 허가한 CPU의 비트마스크를 지정한다. 80x86 구조에서 최대처리기개수는 32므로 전체 마스크를 정수마당 하나에 집어 넣을수 있다.

CPU_rennable

프로쎄스를 실행중인 CPU가 있다면 이것의 비트마스크이다. 프로쎄스를 실행중인 CPU가 없다면 이 마당의 모든 비트를 1로 설정한다. 있다면 프로쎄스를 실행중인 CPU에 해당하는 비트를 1로 설정하고 나머지 비트는 모두 0으로 설정한다. 이런 식으로 지정하면 핵심부는 이 마당과 CPUs_allowed마당, CPU를 지정하는 비트마스크를 론리AND연산을 하여 간단하게 해당 프로쎄스를 지정한 CPU에서 실행하도록 순서짜기할수 있는지 확인할수 있다.

processor

프로쎄스를 실행중인 CPU가 있다면 이것의 색인값이다. 없다면 프로쎄스를 실행한 마지막 CPU의 색인값이다.

do_fork()함수는 새로운 프로쎄스를 생성할 때 current(부모)와 p(자식)가 가리키는 프로쎄스의 counter마당을 다음과 같이 설정한다.

```
p->counter = (current->counter +1) >> 1;
current->counter >>= 1;
if (!current->counter)
    current->need_resched = 1;
```

다시 말해서 부모에 남은 tick수를 반으로 나누어서 절반은 부모에 남겨놓고 나머지 절만은 자식에게 준다. 이렇게 하는 리유는 사용자가 다음과 같은 방법으로 CPU시간을 무제한으로 사용하는것을 막기 위해서이다. 부모프로쎄스가 똑같은 코드를 실행하는 자식프로쎄스를 만들고 자기는 완료한다. 여기에서 프로쎄스를 생성하는 주기를 적당히 조절하면 자식프로쎄스가 부모의 정량이 만료되기전에 항상 새로운 정량을 얻을수 있을것이다. 하지만 이런 프로그람작성법은 핵심부가 생성된 프로쎄스에 기본시간정량을 그대로 주지 않기때문에 동작하지 않는다. 비슷하게 쉘에서 배경프로쎄스를 많이 시작하거나 도형탁상환경에서 창을 많이 열어 불공평하게 자원을 많이 차지하지 못하게 한다. 좀 더 일반적으로 말하면 프로쎄스는 자식을 여러개 만들어서 자원을 독차지할수 없다.(자기에게 실시간 순서짜기방책을 부여할수 있는 권한이 없는 한)

CPU별 자료구조

각 프로쎄스서술자에 들어있는 마당외에도 매 CPU가 무엇을 하고있는지 서술하는

정보가 추가로 필요하다. 이런 목적으로 핵심부는 schedule_data형 NR_CPUS배렬인 aligned_data를 사용한다. 이 구조체는 다음 두 마당으로 이루어진다.

curr

해당 CPU에서 실행중인 프로세스의 프로세스서술자를 가리키는 지적자이다. 보통 이 마당을 접근할 때에는 CPU론리번호 n을 변수로 하여 CPU_curr(n) 마크로를 사용한다.

last_schedule

해당 CPU에서 마지막으로 프로세스절환을 수행한 때의 64bit시간형(Time Stamp Counter)값이다. 보통 이 마당을 접근할 때에는 CPU론리번호 n을 변수로 하여 last_schedule(n)마크로를 사용한다.

대부분의 경우 모든 CPU는 배렬에 있는 자신만의 요소를 접근하기때문에 aligned_data배렬의 모든 요소가 서로 다른 캐쉬에 들어가도록 배치하는것이 편리하다. 이렇게 하면 CPU가 하드웨어캐쉬에 자신의 요소를 포함할 가능성이 높아진다.

2) schedule()함수

schedule()함수는 순서짜기프로그람(scheduler)을 실현한다.

이 함수는 실행대기렬목록에서 실행할 프로세스를 찾아 CPU를 할당한다. 여러 핵심부루틴에서 이 함수를 직접적으로 또는 간접적으로 호출한다.

직접적인 호출

current프로세스가 필요로 하는 자원을 사용할수 없어 이 프로세스를 당장 차단해야 하는 경우에 순서짜기프로그람을 직접호출(direct invocation)한다. 이 경우 프로세스를 차단하려는 핵심부루틴은 다음과 같은 단계를 거친다.

1. current를 적당한 대기렬(wait queue)에 넣는다.
2. current의 상태를 TASK_INTERRUPTIBLE이나 TASK_UNINTERRUPTIBLE중 하나로 바꾼다.
3. schedule()을 호출한다.
4. 자원을 사용할수 있는지 검사하여 사용할수 없으면 2단계로 되돌아간다.
5. 자원을 사용할수 있으면 current를 대기렬에서 제거한다.

보다싶이 핵심부루틴은 프로세스에 필요한 자원이 사용가능한지 반복하여 검사한다. 자원을 사용할수 없으면 schedule()을 호출하여 CPU를 다른 프로세스에게 양보한다. 후에 순서짜기프로그람이 다시 해당 프로세스에 CPU를 할당하면 자원의 사용가능여부를 다시 검사한다.

오랜시간 반복작업을 하는 많은 장치구동프로그람은 순서짜기프로그람을 직접호출하기도 한다. 구동프로그람이 매 반복주기마다 need_resched마당을 검사하고 필요한 경우 schedule()함수를 호출하여 CPU를 자진해서 반납한다.

간접적인 호출

current의 need_resched마당을 1로 설정하여 순서짜기프로그람을 간접적으로 호출(lazy invocation)할수도 있다. 사용자방식프로쎄스로 되돌아가기 전에 항상 이 마당값을 검사하므로 함수는 가까운 시간안에 반드시 호출된다.

례를 들어 다음과 같은 경우에 순서짜기프로그람을 우회적으로 호출한다.

· current가 자신의 CPU시간의 정량을 모두 사용했을 때. update_process_times()함수가 이것을 수행한다.

· 어떤 프로쎄스가 깨여났는데 그 프로쎄스의 우선순위가 현재프로쎄스의 우선순위보다 높을 때. 주로 wake_up_process()함수가 호출하는 reschedule_idle()함수가 이것을 수행한다.(앞절의 《프로쎄스구별하기》참고)

· sched_setscheduler() 또는 sched_yild()체계호출을 실행할 때.(뒤에 나오는 《순서짜기관련체계호출》참고)

프로쎄스절환전에서 schedule()함수가 수행하는 일

schedule()함수의 목적은 현재 실행중인 프로쎄스를 다른 프로쎄스로 교체하는것이다. 따라서 이 함수의 핵심결과물은 currnet프로쎄스를 대신할 프로쎄스를 선택하여 next라는 변수를 이 프로쎄스의 서술자에 대한 지적자로 설정하는것이다. 체계에 current보다 우선순위가 높은 실행가능한 프로쎄스가 없다면 next는 current와 일치하고 따라서 프로쎄스절환은 일어나지 않는다.

schedule()함수는 효률성을 위해 변수 몇개를 초기화하며 시작한다.

```
prev = current;
this_CPU = prev->processor;
sched_data = &aligned_data[this_CPU];
```

여기서 보는것처럼 current가 반환하는 지적자를 prev에 저장하고 현재 실행중인 CPU의 론리번호를 this_CPU에 저장하며 aligned_data배렬에서 이 CPU에 해당하는 항목에 대한 지적자를 sched_data에 저장한다.

다음으로 prev가 대역핵심부잠그기(global kernel lock)나 대역새치기잠그기(global interrupt lock)를 담지 않게 하고(《대역핵심부잠그기》와 《대역새치기금지》 참고) 새치기를 다시 허용한다.

```
if (prev->lock_depth >= 0)
spin_unlock (&kernel_flag) ;
release_irqlock (this_CPU) ;
__sti ( ) ;
```

일반적으로 프로쎄스는 프로쎄스절환을 거치는 동안 잠그기를 소유하면 안된다. 잠그기를 소유하면 다른 프로쎄스가 같은 잠그기를 획득하려는 즉시 체계가 멈추게 된다. 그렇지만 schedule()함수가 lock_depth마당의 값을 바꾸지 않는데 주목하자. prev가 실행을 재개할 때 이 마당의 값이 부수가 아니라면 kernel_flag스핀잠그기를 다시 획득

한다. 따라서 프로세스절환을 거치면서 대역핵심부잠그기를 자동으로 해제하고 다시 획득하게 된다. 반면에 대역새치기잠그기는 자동으로 다시 획득하지 않는다.

schedule()은 실행가능한 프로세스들을 들여다보기 시작하기 전에 지역새치기를 금지하고 실행대기렬을 보호하는 스핀잠그기를 획득해야 한다.

```
spin_lock_irq(&runqueue_lock);
```

다음으로 prev가 주어진 정량을 모두 사용한 Round Robin방식의 실시간프로세스인지(policy마당이 SCHED_RR) 검사한다. 옳다면 schedule()함수는 prev에 새 정량을 할당하고 이것을 실행대기렬목록의 맨끝에 넣는다.

```
if (prev->policy == SCHED_RR && !prev->counter) {
prev->counter = (20- prev->nice) / 4 + 1;
move_last_runqueue(prev);
}
```

프로세스의 nice마당은 -20에서 +19사이의 범위에 사실을 상기하자. 따라서 schedule()은 counter마당을 11에서 1사이의 tick수로 다시 채운다. nice마당의 기본값은 0이므로 일반적으로 프로세스는 6tick, 즉 대략 60ms의 새로운 정량을 소유한다.

다음으로 schedule()함수는 prev의 상태를 검사한다. 프로세스에 차단하지 않는 대기중인 신호가 있으면서 프로세스상태가 TASK_INTERRUPTIBLE이라면 프로세스상태를 TASK_RUNNING으로 설정한다. 이 작업은 prev에 처리기를 할당하는것이 아닌 단지 prev에 실행할 프로세스로 선택될수 있는 기회를 주는것뿐이다.

```
if (prev->state == TASK_INTERRUPTIBLE && signal_pending(prev))
prev->state = TASK_RUNNING;
```

prev가 TASK_RUNNING 상태가 아니라면 프로세스가 어떤 외부자원을 기다려야해서 프로세스가 직접 schedule()함수를 호출한 경우이다. 따라서 실행대기렬목록에서 prev를 제거해야 한다.

```
 if (prev->state != TASK_RUNNING)
   del_from_runqueue (prev) ;
```

schedule()함수는 순서짜기프로그람을 간접적인 방법으로 실행한 경우 current 의 need_resched마당을 0으로 재설정한다.

```
prev->need_resched =0;
```

이제 schedule()함수는 다음 시간정량동안 실행할 프로세스프레임 선택할 차례이다. 이를 위해 함수는 실행대기렬목록을 검색한다. 이 코드의 목적은 next에 우선순위가 가장 높은 처리기의 프로세스서술자지적자를 저장하는것이다.

```
repeat_schedule:
   next = init_tasks[this_CPU] ;
c = -1000;
```

```
list_for_each (tmp, &runqueue_head) {
        p = list_entry(tmp, struct task_struct, run_list);
        if (p->CPUs_runnable & p->CPUs_allowed & (1<<this_CPU) ) {
                int weight = goodness(p, this_CPU, prev->active_mm);
                if(weight > c)
                        c = weight, next = p;
        }
}
```

이 함수는 init_task[this_CPU]가 가리키는 프로세스 즉 실행중인 CPU에 관련된 프로세스 0을 가리키는 지적자를 저장하도록 next를 초기화한다. 그리고 변수 c를 -1000으로 설정한다. 후에 《실행가능한 프로세스가 얼마나 우수한가》에서 보지만 goodness()함수는 변수로 전달한 프로세스의 우선순위를 나타내는 옹근수를 반환한다.

실행대기렬에 있는 프로세스를 검색할 때 schedule()은 다음 두 조건 모두에 해당하는 프로세스만을 고려한다.

1. 실행중인 CPU에서 실행가능하다.(cpus_allowed & (1 << this_cpu))
2. 다른 CPU에서 실행중이 아니다.(cpus_runnable & (1<< this_cpu) 앞에서 cpus_runnable에 관한 설명 참고)

이 순환은 실행대기렬에서 무게값이 가장 높은 첫번째 프로세스를 선택한다. 따라서 검색을 마치면 next는 가장 우수한 후보를 가리키고 변수 c는 해당 프로세스의 우선순위를 저장한다. 실행대기렬이 비여있을수도 있는데 이때에는 순환을 하지 않고 next는 실행중인 CPU와 관련된 교환기핵심부스레드를 가리킨다. 또한 가장 우수한 후보가 이전 현재프로세스인 prev일수도 있다. 순환을 빠져나갈 때 변수 c가 0 값인 특수한 경우가 있다. 이것은 현재 실행중인 CPU에서 실행할수 있는 실행대기렬목록에 있는 모든 프로세스가 자신의 정량을 완전히 소비했을 때 즉 해당 프로세스들의 counter마당값이 모두 0인 경우이다. 이때에는 새로운 시기를 시작해야 한다. schedule()함수는 존재하는 모든 프로세스에(TASK_RUNNING상태인 프로세스뿐만아니라) 새로운 정량으로 counter값의 절반에 nice값에 따른 증가값을 더한 값을 부여한다.

```
if (!c) {
struct task_struct *p;
   spin_unlock_irq(&runqueue_lock) ;
read_lock(&tasklist_lock) ;
for_each_task(p)
        p->counter = (p->counter >> 1) + (20 - p->nice) / 4 + 1;
read_unlock(&tasklist_lock) ;
spin_lock_irq(&runqueue_lock) ;
```

```
goto repeat_schedule;
}
```

이렇게 해서 보류되거나(suspended) 중지된(stopped) 프로쎄스의 동적우선순위는 주기적으로 높아진다. 앞에서 설명한대로 보류되거나 중지된 프로쎄스의 counter 값을 증가시키는것은 입출력위주 프로쎄스를 선취하도록 하기 위해서이다. 그렇지만 정량이 자주 증가하더라도 그 값은 절대로 약 230ms를 넘을수 없다.

이제 schedule()함수가 최우수후보를 선택해서 next가 그 프로쎄스서술자를 가리킨다고 하자. 다음으로 aligned_data배렬에서 실행중인 CPU에 해당하는 요소를 갱신하고(변수 sched_data가 이 요소를 참조한다.), next의 프로쎄스서술자에 있는 실행중인 CPU의 색인값을 기록하고 실행대기렬목록의 스핀잠그기를 해제하고 국부새치기를 다시 허용한다.

```
sched_data->curr = next;
next->processor = this_cpu;
next->cpus_runnable = 1UL << this_cpu;
spin_unlock_irq(&runqueue_lock) ;
```

이제 schedule()함수는 실제 프로쎄스절환을 진행할 준비가 되였다. 그러나 먼저 한가지 알아둘 사항이 있다. 최우수후보인 next가 이전에 실행하던 프로쎄스 prev와 같다면 schedule()함수를 끝마쳐도 된다.

```
if (prev == next) {
prev->policy &= ~SCHED_YIELD;
if (prev->lock_depth >= 0)
        spin_lock (&kernel_flag) ;
return;
}
```

schedule()함수는 프로쎄스의 lock_depth마당이 부수가 아니라면 대역핵심부잠그기를 다시 획득한다는점에 류의하자. 이것은 이 함수의 첫번째 작업을 설명할 때 언급하였다.

prev가 아닌 다른 프로쎄스를 선택하였다면 프로쎄스절환이 일어나야 한다. rdtsc 기호언어명령을 사용하여 시간형계수기의 현재값을 알아내서 이것을 aligned_data배렬에서 실행중인 CPU에 해당하는 항목의 last_schedule 마당에 저장한다.

```
asm volatile ("rdtsc" : “=A” (sched_data->last_schedule));
```

kstat의 context_swtch마당을 1증가시켜 핵심부가 관리하는 통계정보를 갱신한다.

```
kstat.context_swtch++ ;
```

next의 주소공간을 옳바로 설정하는것 역시 필수적이다. 앞절에서 프로쎄스서술자의 active_mm마당은 프로쎄스가 실제로 사용하는 기억기서술자를 가리키고 mm마당은

프로쎄스가 소유하는 기억기서술자를 가리킨다고 서술하였다. 보통 프로쎄스에서 두 마당의 주소는 같지만 핵심부스레드는 자신의 주소공간이 없으므로 mm마당은 항상 NULL값이다. schedule()함수는 next가 핵심부스레드라면 prev가 사용하던 주소공간을 사용하게 만든다.

```
if ( !next->mm) {
next->active_mrn = prev->active_mm;
atomic_inc(&prev->active_mm->mm_count) ;
cpu_tlbstate[this_cpu].state = TLBSTATE_LAZY;
}
```

이전 판본의 Linux에서 핵심부스레드는 자기만의 주소공간이 있었다. 순서짜기프로그람이 새로 실행할 프로쎄스로 핵심부스레드를 선택한 경우 페지표를 교체하는것은 쓸모없으므로 이런 설계는 최선은 아니였다. 핵심부방식에서 동작하는 핵심부스레드는 선형주소공간의 마지막 1GB만을 사용하며 체계의 모든 프로쎄스에서 이 령역에 대한 사용은 똑같다. 더 나쁜점은 cr3등록기에 쓰기를 하면 모든 TLB입구점을 무효화해서 성능에 큰 손실을 준다. Linux 2.6은 next가 핵심부스레드라면 페지표를 건드리지 않으므로 더 효률적이다. 보다 더 최적화하기 위해서 next가 핵심부스레드인 경우 schedule()함수는 프로쎄스를 지연TLB방식으로 설정한다.

반면에 next가 정규프로쎄스라면 schedule()함수는 prev의 주소공간을 next의 주소공간으로 교체한다.

```
if (next->mm)
switch_mm(prev->active_mm, next->mm, next, this_cpu) ;
```

prev가 핵심부스레드라면 schedule()함수는 prev가 사용하던 주소공간을 해제하고 prev->active_mm을 재설정한다.

```
if ( !prev->mm) {
mmdrop(prev->active_mm) {
prev->active_mm = NULL;
}
```

mmdrop()는 기억기서술자의 사용회수를 감소시킨다는것을 상기하자. 이 회수가 0이 되면 이 서술자와 이와 관련된 페지표와 가상기억기령역도 해제한다.

이제 schedule()은 마지막으로 switch_to()를 호출하여 prev와 next간에 프로쎄스절환을 수행한다.

```
switch_to(prev, next, prev);
```

프로쎄스절환후에 schedule()이 하는 일

schedule()함수에서 switch_to마크로를 호출하는 명령이후에 나오는 명령은 next 프로쎄스가 실행하는것이 아니고 후에 순서짜기프로그람이 prev를 다시 선택하여 실행

하려고 할 때 prev프로쎄스가 실행한다. 그런데 그 당시의 변수 prev는 우리가 schedule()함수를 설명하기 시작할 때 교체되여 나간 원래 프로쎄스를 가리키는것이 아니라 prev를 다시 실행하도록 순서짜기할 때 이 프로쎄스에 교체당하는 프로쎄스를 가리킨다.

schedule()함수의 마지막명령은 다음과 같다.

```
__schedule_tail(prev) ;
if (current->look_depth >= 0)
spin_lock (&kernel_flag);
if (current->need_resched)
goto need resched_back;
return ;
```

보다싶이 schedule()은 __schedule_tail()을 호출하고 필요하다면 대역핵심부잠그기를 다시 획득하고 다른 프로쎄스가 현재프로쎄스의 need_resched마당을 설정했는가를 검사한다. 이 경우 전체 schedule()함수를 처음부터 다시 시작하고 그렇지 않으면 완료한다.

단일처리기체계에서 __schedule_tail()함수는 prev의 policy마당에 있는 SCHED_YIELD기발을 지우는 일만 한다. 반면에 다중처리기체계에서 이 함수는 본질적으로 다음 코드쪼각과 등등한 코드를 실행한다.

```
policy = prev->policy;
prev->policy = policy & -SCHED_YIELD;
wmb( ) ;
spin_lock(&prev->alloc_lock) ;
prev->cpus_runnable = ~0UL;
spin_lock_irqsave(&runqueue_lock, flags);
if (prev->state == TASK_RUNNING && prev != init_task[smp_processor_id
()]
        && prev->cpus_runnable == ~0UL && !(policy & SCHED_YIELD))
   reschedule_idle(prev);
spin_unlock_irqrestore(&runqueue_lock, flags);
spin_unlock(&prev->alloc_lock);
```

처리기가 policy마당을 갱신하는 기호언어명령과 alloc_lock스핀잠그기를 획득하는 명령을 뒤섞어버리지 않도록 wmb()기억기장벽을 사용한다. 다중처리기체계에서 __schedule_tail()의 역할은 상당히 중요하다. 이 함수는 교체되여 나간 프로쎄스를 다른 CPU에서 다시 순서짜기할수 있는가를 검사하기때문이다. 다음과 같은 조건을 충족하는 경우에만 이런 시도를 한다.

- prev는 TASK_RUNNING상태이다.
- prev는 실행하는 CPU의 교환기프로쎄스가 아니다.
- prev->policy의 SCHED_YEILD기발이 설정되여있지 않다.
- cpus_runnable마당을 설정한 때부터 if문장에 이르는 시간동안(if문장은 runqueue_lock스핀잠그기로 보호하고있다.) 다른 CPU가 prev를 선택하지 않았다.

__schedule_tail()은 prev의 우선순위가 다른 CPU의 현재프로쎄스를 교체할만한가를 검사하려고 reschedule_idle()을 호출한다. wake_up_process()도 이 함수를 호출하며 후에 《다중처리기체계에서 순서짜기》에서 설명한다. 다음 두 부분을 통해 순서짜기프로그람분석을 완료한다. 이것은 각각 goodness()함수와 reschedule_idle()함수를 설명한다.

3) 실행가능한 프로쎄스가 얼마나 우수한가

순서짜기알고리듬의 핵심은 실행대기렬목록에 있는 모든 프로쎄스중에서 가장 우수한 후보를 찾는것이다. 이 작업이 바로 goodness()함수가 하는 일이다. 이 함수는 다음과 같은 변수를 받는다.

- 후보프로쎄스의 서술자를 가리키는 지적자 p
- 실행중인 CPU의 론리번호 this_cpu
- 교체해 나갈 프로쎄스의 기억기서술자의 주소 this_mm

goodness()함수가 반환하는 정수값 weight는 p의 《우수성》을 측정한 값으로서 다음과 같은 의미를 가진다.

weight = -1

p는 prev프로쎄스이고 SCHED_YIELD기발이 1이다. 실행대기렬에 실행가능한 다른 프로쎄스(교환기프로쎄스를 제외하고)가 없는 경우에만 이 프로쎄스를 선택한다.

weight = 0

자기의 정량을 모두 소비한(p->counter가 0) 일반프로쎄스이다. 실행가능한 모든 프로쎄스가 자기의 정량을 완전히 소비한 경우가 아니라면 이 프로쎄스를 실행할 프로쎄스로 선택하지 않는다.

2 < weight <= 77

p는 정량을 다 소비하지 않은 일반프로쎄스이다. 다음과 같이 무게값을 계산한다.

```
weight = p->counter + 20    p->nice ;
if (p -> processor == this_cpu)
        weight += 15;
if (p->mm == this_mrn || !p->mm)
        weight += 1;
```

다중처리기체계에서는 해당 프로쎄스를 마지막으로 실행한 처리기가 순서짜기프

로그람을 실행하는 프로쎄스라면 아주 큰 값(+15)을 준다. 이것은 프로쎄스가 여러 CPU사이를 오가는 회수를 줄여서 하드웨어캐쉬놓침회수를 줄이는데 도움을 준다.

이 함수는 프로쎄스가 핵심부스레드이거나 이전에 실행하던 프로쎄스와 기억기주소공간을 공유하면 약간의 값(+1)을 준다. 주로 cr3등록기에 쓰기를 하여 TLB를 무효화하지 않으려고 이런 프로쎄스를 선취한다.

weight >= 1000

p는 실시간프로쎄스이다. weight는 p->counter + 1000값이다.

4) 다중처리기체계에서 순서짜기

Linux2.6에서는 이전판본과 비교하여 다중처리기체계에서의 성능이 향상되도록 순서짜기알고리듬을 개선하였다. 또한 순서짜기프로그람을 단순화했는데 이 자체로도 큰 개선이다.

지금까지 본것처럼 매 처리기는 자기가 현재 실행중인 프로쎄스를 교체하려고 schdule()함수를 실행한다. 그렇지만 체계성능을 높이려고 처리기끼리 정보를 교환하는것도 가능하다. 특히 프로쎄스절환직후 어떤 처리기든지 대체로 방금 교체된 프로쎄스를 이보다 우선순위가 낮은 프로쎄스를 실행하는 다른 CPU에서 실행해야 하는가를 검사한다. 이 작업이 reschedule_idle()에서 수행하는 일이다.

reschedule_idle()함수는 변수로 전달한 프로쎄스 p를 실행할 다른 CPU를 찾아 처리기간새치기를 사용하여 다른 CPU가 순서짜기를 수행하게 만든다. 이 함수는 정해진 순서대로 일련의 검사를 수행한다. 그 가운데서 하나라도 성공하면 선택한 CPU에 RESCHEDULE_VECTOR처리기간 새치기프레임전송하고 완료한다. 검사가 실패하면 다시 순서짜기를 하지 않고 완료한다. 검사하는 순서는 다음과 같다.

1. p를 마지막으로 실행한 CPU(즉 p->processor색인값에 해당하는 CPU)가 쉬고있는가

```
best_cpu = p->processor;
if ((p->cpus_allowed & p->cpus_runnable & (1 << best_cpu))
&& cpu_curr(best_cpu) == init_tasks[best_cpu]) {
send_now_idle:
need_resched= init_tasks[best_cpu] ->need_resched;
init_tasks[best_cpu] ->need_resched = 1;
if (best_cpu != smp_processor_id( ) && !need_resched)
        smp_send_reschedule(best_cpu) ;
}
```

이 경우 프로쎄스를 선취할 필요가 없고 해당 처리기의 하드웨어캐쉬가 여전히 쓰이기때문에(유용한 자료가 들어있음) 가장 좋은 경우이다. schedule()함수가 실행가능한

교육성 교육정보쎈터

프로쎄스를 좇아내고 이것을 교환기핵심부스레드로 교체하는 일은 없으므로 순서짜기프로그람이 reschedule_idle()을 호출할 때에는 이런 경우가 일어나지 않는다. 그러나 wakeup_process()가 reschedule_idle()을 호출할 때 즉 p가 방금 막 깨여났을 때에는 이런 경우가 있을수 있다.

대상처리기에서 다시 순서짜기가 일어나게 하려고 교환기핵심부스레드의 need_resched마당을 1로 설정한다. 대상처리기가 reschedle_idle()함수를 실행하는 처리기와 다르다면 RESCHEDULE_VECTOR처리기간 새치기도 발생시킨다. 사실 쉬고있는 처리기는 일반적으로 전력소비를 줄이려고 halt기호언어명령을 실행하므로 새치기만이 해당 CPU를 깨울수 있다. 그렇지만 다시 순서짜기하는 속도를 빠르게 하고 처리기간 새치기를 사용하지 않기 위해 교환기핵심부스레드가 능동적으로 need_resched마당을 계속해서 검사하면서 그 값이 -1에서 +1로 바뀌길 기다릴수도 있다. 기동단계에서 핵심부에 《idle=poll》변수를 전달하여 이러한 전력을 많이 소비하는 알고리듬을 활성화할수 있다.

2. p를 실행할수 있는 쉬고있는 처리기가 있는가

```
oldest_idle = -1;
for (cpu= 0; cpu<smp_num_cpus, cpu++) {
        if (!(p->cpus_allowed & p->cpus_runnable & (1 << cpu)))
            continue;
        if (cpu_curr (cpu) == init_tasks [cpu]
                && last_schedule (cpu) < oldest_idle)
            oldest idle = last schedule (cpu) ;
                target_tsk = cpu_curr (cpu) ;
}
if (oldest_idle != -1) {
        best_cpu = target_tsk->processor;
         goto send_now_idle;
   }
```

이 함수는 p를 실행할수 있는 쉬고있는 처리기중 가장 오래동안 사용하지 않은 처리기를 선택한다. 모든 CPU에서 가장 마지막으로 프로쎄스절환이 일어난 시점의 시간형계수기를 aligned_data배렬에 저장한다는것을 상기하자. 이 함수는 가장 오래동안 쉬고있는 CPU를 찾아서 1번에서 설명한 재순서짜기를 일으키는 건너뛴다. 최장시간휴식규칙(oldest idle rule)의 리론적인 근거는 이 CPU가 가장 많은 수의 무효한 하드웨어캐쉬행을 가지는 경향이 있다는것이다.

3. p를 실행할수 있으면서 현재프로쎄스의 동적우선순위가 p보다 낮은 프로쎄스가 있는가

```
max_prio = 0;
for (cpu = 0; cpu<smp_num_cpus; cpu++) {
  if ( !(p->cpus_allowed & p->cpus_runnable & (1 << cpu) ) )
        continue;
prio= goodness (p, cpu, cpu_curr (cpu) ->active_mm) -
        goodness (cpu_curr (cpu) , cpu, cpu_curr (cpu) ->active_mm);
          if (prio > max_prio)
        max_prio = prio, target_tsk = cpu_curr (cpu) ;
        }
if (max_prio > 0) {
        target_tsk->need_resched = 1;
          if (target_tsk->processor != smp_processor_id())
                smp_send_reschedule(target_tsk->processor);
   }
```

reschedule_idle()은 현재프로쎄스를 p로 교체할 때의 좋은 정도와 현재프로쎄스를 현재프로쎄스자기로 교체할 때의 좋은 정도의 차이가 가장 큰 처리기를 찾는다. 최대값이 정수면 해당 처리기에서 재순서짜기가 일어나게 만든다. 이 함수는 단순히 해당 프로쎄스의 counter와 nice마당만 보는것이 아니고 현재 실행중인 프로쎄스를 다른 주소공간을 사용할수 있는 다른 프로쎄스로 교체할 때 드는 비용까지 고려하는 goodness() 함수를 사용한다.

5) 순서짜기알고리듬성능

Linux순서짜기알고리듬은 자체로 필요한 기능을 내장하고있으며 상대적으로 따라가기도 쉽다. 따라서 많은 핵심부해커가 이것을 더욱 개선하려고 시도한다. 그렇지만 순서짜기프로그람은 핵심부중에서도 아주 신비한 부분이다. 몇가지 핵심변수를 바꿔서 성능을 크게 바꿀수 있지만 대부분 얻은 결과를 뒤받침할만한 리론적인 기반이 없다. 더구나 이렇게 얻은 긍정적인(또는 부정적인) 결과가 다양한 사용자요청(실시간, 호상작용, 입출력위주, 배경작업 등)을 섞는 방법을 크게 바꾼 경우에도 그대로 지속되리라 확신할수도 없다. 실제로 제안한 순서짜기방책의 대부분은 요청을 인위적으로 혼합하여 낮은 체계성능을 내도록 하는것이 가능하다.

Linux2.6 순서짜기프로그람에 있는 몇가지 함정을 간단히 정리해보자. 후에 나오겠지만 여기서 설명하는 제한사항중 몇가지는 많은 사용자를 수용하는 대형체계에서 아주 중요하다. Linux순서짜기프로그람은 Workstation한대에서 동시에 프로쎄스 수십개를 실행하는 경우에 무척 효률적이다.

알고리듬은 규모가 커지면 잘 동작하지 않을수 있다

존재하는 프로쎄스수가 매우 많다면 한번에 모든 동적우선순위를 다시 계산하는것은 비효률적이다.

전통적인 Unix핵심부에서는 매초마다 동적우선순위를 다시 계산해서 문제가 더욱 심각하였다. 그대신 Linux는 순서짜기프로그람의 부하를 최소화하기 위해 노력한다. Linux는 모든 실행가능한 프로쎄스가 자신의 시간정량을 모두 소비했을 때에만 우선순위를 다시 계산한다. 따라서 프로쎄스수가 많아지면 다시 계산하는데 시간이 더 걸리겠지만 재계산회수는 줄어든다.

이런 간단한 접근방법에는 부족점이 있다. 실행가능한 프로쎄스수가 아주 많고 입출력위주프로쎄스가 가끔씩 실행되는 경우에는 호상작용을 하는 응용프로그람의 반응시간은 더 길어진다.

체계에 부하가 많이 걸리는 경우 정해진 정량기간은 너무 길다

사용자가 느끼는 체계반응성은 실행가능한 즉 CPU시간을 할당받으려고 기다리는 프로쎄스의 평균개수인 체계부하(system load)에 따라 달라진다.

앞에서 언급한대로 체계반응성은 실행가능한 프로쎄스의 평균시간정량기간에도 의존한다. Linux에서 미리 정의한 시간정량값은 체계부하가 매우 높은 고성능체계에는 아주 큰 편이다.

입출력위주프로쎄스를 선취하는것이 최선은 아니다

입출력위주프로쎄스를 선취하는것은 호상작용하는 프로그람의 반응시간을 줄이는 좋은 방도이지만 완벽하지는 않다. 실제로 거의 사용자와 호상작용을 하지 않는 일괄작업프로그람에서도 입출력위주인 프로그람이 있다. 례를 들어 하드디스크에서 많은 자료를 읽어야 하는 자료기지검색엔진이나 저속망련결을 통해서 원격주콤퓨터에서 자료를 수집해야 하는 망응용프로그람을 생각해보자. 이런 종류의 프로쎄스는 반응시간이 짧을 필요는 없지만 순서짜기알고리듬은 이것들을 선취한다. 반면에 CPU위주의 호상작용프로그람인 경우 CPU를 사용하면서 우선순위가 낮아지는데 차단을 일으키는 입출력연산때문에 높아지는 동적우선순위가 이에 미치지 못하므로 사용자에게는 반응성이 떨어져보일수도 있다.

실시간응용프로그람지원이 취약하다

1장에서 설명한것처럼 비선취형핵심부는 실시간응용프로그람에 적당하지 않다. 프로쎄스가 새치기나 례외를 처리하느라 핵심부방식에서 몇미리초를 보낼수도 있기때문이다. 이 시간동안 실행가능해진 실시간프로쎄스의 실행을 재개할수 없다. 반응시간이 예측가능하고 짧아야 하는 실시간응용프로그람은 이것을 받아들일수 없다.

앞으로 등장할 Linux판본은 SVR4의 고정선취지점(fixed preemption point)을 실현하거나 핵심부를 완전히 선취가능하게 하여 이 문제를 해결할수도 있을것이다. 그렇지만 이런 설계가 Linux같은 범용조작체계에 적당한가는 의문으로 남아있다.

사실 핵심부선취은 효률적인 실시간순서짜기프로그람을 실현하기 위한 몇가지 필수

조건중 하나일뿐이다. 이밖에도 고려해야 하는 여러 문제가 있다. 례를 들어 실시간프로쎄스는 종종 일반프로쎄스가 사용중인 자원을 필요로 한다. 그래서 실시간프로쎄스는 우선순위가 자신보다 낮은 프로쎄스가 자원을 해제할 때까지 기다려야 할수도 있다. 이런 현상을 우선순위반전(priority inversion)이라고 한다. 게다가 실시간프로쎄스는 다른 우선순위가 낮은 프로쎄스(례를 들면 핵심부스레드)에 맡긴 핵심부봉사를 필요로 할수도 있다. 이런 현상을 가리켜 숨은 순서짜기(hidden scheduling)라고 한다. 효률적인 실시간순서짜기는 이런 문제를 고려하고 해결해야 한다.

현재 이런 모든 결점을 고친 새로운 순서짜기프로그람을 개발하여 Linux 2.6에 추가하였다. 이 순서짜기프로그람은 매우 효률적이다.

3. 순서짜기관련체계호출

프로쎄스가 자신의 우선순위와 순서짜기방책을 바꿀수 있도록 여러가지 체계호출을 도입하였다. 일반적인 규칙은 사용자는 자신이 소유한 프로쎄스의 우선순위를 언제든지 낮출수 있다는것이다. 그러나 다른 사용자가 소유한 프로쎄스의 우선순위를 바꾸려 하거나 자신이 소유한 프로쎄스의 우선순위를 높이려 한다면 반드시 관리자권한이 있어야 한다.

1) nice()체계호출

nice()체계호출은 프로쎄스가 자신의 기본우선순위를 바꿀수 있게 한다. increment변수로 전달한 정수값을 프로쎄스서술자의 nice마당을 바꾸는데 사용한다. Unix명령 nice는 사용자가 순서짜기우선순위를 바꿔서 프로그람을 실행할수 있도록 이 체계호출을 리용한다.

sys_nice()봉사루틴은 nice()체계호출을 처리한다.

increment변수에 어떤 값이든 지정할수 있지만 절대값이 40을 넘어가면 이것을 40으로 맞춘다. 부수는 우선순위를 높이는 요청으로 관리자권한이 필요하다. 반면에 정수는 우선순위를 낮추는 요청이다. 부수인 경우 이 함수는 capable()함수를 호출하여 프로쎄스에 CAP_SYS_NICE 특질(capability)이 있는가를 확인한다. capable()함수와 특질의 개념은 후에 설명한다.

사용자가 우선순위를 바꾸는데 필요한 특질이 있음을 확인하면 sys_nice()는 current의 nice마당에 increment값을 더한다. 필요하다면 이 마당의 값을 조정하여 그 값이 -20보다 작거나 19보다 클수 없게 만든다.

이제는 호환성을 위해서만 nice()체계호출을 유지하며 다음에 설명하는 setpriority()체계호출이 이것을 대신한다.

2) getpriority()와 setpriority() 체계호출

nice()체계호출은 이것을 호출한 프로쎄스에만 영향을 미친다. getpriority()와

setpriority()체계호출은 지정한 프로쎄스그룹에 있는 모든 프로쎄스의 기본우선순위에 영향을 미친다. getpriority()는 20에서 지정한 그룹내에 있는 모든 프로쎄스의 기본우선순위중 가장 낮은 nice마당의 값을 뺀 값을 반환한다. setpriority()는 지정한 그룹내에 있는 프로쎄스의 기본우선순위를 지정한 값으로 설정한다.

핵심부는 이 체계호출을 각각 sys_getpriority()와 sys_setpriority() 봉사루틴으로 실현한다.(kernel\sys.c)

이 두 함수는 모두 다음과 같은 변수를 사용한다.

which

프로쎄스그룹을 식별한다. 다음 값중 하나이다.

PRIO_PROCCESS

프로쎄스 ID(프로쎄스서술자의 pid마당)와 일치하는 프로쎄스를 선택한다.

PRIO_PGRP

그룹 ID(프로쎄스서술자의 pgrp마당)와 일치하는 프로쎄스를 선택한다.

PRIO_USER

사용자 ID(프로쎄스서술자의 uid마당)와 일치하는 프로쎄스를 선택한다.

who

프로쎄스를 선택하는데 사용하는 pid나 pgrp, uid마당(which의 값에 따라)의 값이다. who가 0이면 이 값을 current프로쎄스의 해당 마당값으로 설정한다.

niceval

새로운 기본우선순위값이다.(sys_setpriority()에만 필요하다.) -20(가장 높은 우선순위)에서 +19(가장 낮은 우선순위)사이의 값이다.

앞서 설명한바와 같이 CAP_SYS_NICE특질이 있는 프로쎄스만 자신의 기본우선순위를 높이거나 다른 프로쎄스의 기본 우선순위를 바꿀수 있다.

앞에서 본것처럼 체계호출은 오유가 발생한 경우에만 부수값을 반환한다. 따라서 getpriority()함수는 -20에서 +19사이에 있는 일반 nice값이 아니고 1에서 40사이에 있는 부수가 아닌 값을 반환한다.

3) 실시간프로쎄스관련 체계호출

이제부터 프로쎄스가 자신의 순서짜기규칙을 바꾸는 방법과 특별히 실시간프로쎄스가 되게 하는 일련의 체계호출을 소개한다.

여기서도 마찬가지로 프로쎄스는 자기자신과 다른 프로쎄스의 프로쎄스서술자에 있는 rt_priority와 policy값을 바꾸려면 CAP_SYS_NICE특질이 있어야 한다.

sched_getscheduler()와 sched_setscheduler() 체계호출

sched_getscheduler()체계호출은 pid변수로 지정한 프로쎄스에 현재 적용하는 순서짜기방책을 질문한다. pid가 0이면 이 체계호출을 실행한 프로쎄스의 순서짜기방책을

반환한다. 이 체계호출이 성공하면 지정한 프로세스의 순서짜기방책으로 SCHED_FIFO나 SCHED_RR, SCHED_OTHER를 반환한다. 이 체계호출을 처리하는 sys_sched_getscheduler()봉사루틴은 find_process_by_pid()함수를 호출하여 지정한 pid에 해당하는 프로세스서술자를 찾아 policy마당값을 반환한다.

sched_setscheduler()체계호출은 pid변수로 지정한 프로세스의 순서짜기방책과 이와 관련한 인자를 설정한다. pid가 0이면 이것을 호출한 프로세스의 순서짜기프로그람인자를 설정한다.

해당 봉사루틴인 sys_sched_setscheduler()함수는 policy변수로 지정한 순서짜기방책과 param->sched_priority변수로 지정한 새로운 정적우선순위가 정확한 값인가를 검사한다. 또한 프로세스에 CAP_SYS_NICE특질이 있는지 또는 소유자가 관리자권한을 소유하고있는지 검사한다. 모든것이 정상이라면 다음 코드를 실행한다.

```
p->policy = policy;
p->rt_priority = param->sched_priority;
if (task_on_runqueue (p) )
move_first_runqueue (p) ;
current->need_resched = 1;
```

sched_getparam()과 sched_setparam()체계호출

sched_getparam()체계호출은 pid에 해당하는 프로세스의 순서짜기인자를 얻는다. pid가 0이면 current프로세스의 순서짜기인자를 얻는다. 이것을 처리하는 sys_sched_getparam()봉사루틴은 이미 예상하고있겠지만 pid와 련관된 프로세스서술자의 지적자를 찾아서 프로세스서술자내의 rt_priority마당을 sched_param변수에 저장한다.

그리고나서 copy_to_user()를 호출하여 param변수로 지정한 프로세스주소공간에 있는 주소로 이것을 복사한다. sched_setparam()체계호출은 sched_setscheduler()와 비슷하지만 이 체계호출은 policy마당값을 설정하지 않는다는 차이가 있다. 이것을 처리하는 sys_sched_setparam()봉사루틴은 sys_sched_setscheduler()와 거의 똑같지만 이것을 적용하는 프로세스의 순서짜기방책은 절대 바뀌지 않는다.

sched_yield()체계호출

sched_yield()체계호출을 사용하면 프로세스를 보류상태로 만들지 않고 자발적으로 CPU를 반납할수 있다. 프로세스는 여전히 TASK_RUNNING 상태로 남지만 순서짜기프로그람은 이 프로세스를 실행대기렬목록의 맨 끝에 넣어 동적우선순위가 같은 다른 프로세스에 실행할 기회를 준다. 이 체계호출은 SCHED_FIFO프로세스가 주로 사용한다.

이 체계호출을 처리하는 sys_sched_yield()봉사루틴은 먼저 체계에 이 체계호출을 실행하는 프로세스와 교환기(swapper)핵심부스레드를 제외한 다른 실행가능한 프로세스가 있는가를 검사한다. 그런 프로세스가 없다면 처리기를 놓아주더라도 이것을 사용할

프로쎄스가 없기때문에 sched_yield()는 아무일도 하지 않고 완료한다. 만일 있다면 다음 문장을 실행한다.

```
if (current->policy == SCHED_OTHER)
current->policy != SCHED_YIELD;
current->need_resched = 1;
spin_lock_irq(&runqueue_lock) ;
move_last_runqueue (current) ;
spin_unlock_irq(&runqueue_lock) ;
```

그 결과 sys_sched_yield()봉사루틴을 빠져나갈 때 schedule()함수가 호출되고 (《새치기와 례외로부터 복귀》를 참고) 이것은 대체로 현재프로쎄스를 다른 프로쎄스로 교체한다.

- **sched_get_priority_min()과 sched_get_priority_max()체계호출**

sched_get_priority_min()과 sched_get_priority_max()체계호출은 각각 polcy변수로 지정한 순서짜기방책에서 사용할수 있는 실시간 정적 우선순위의 최소, 최대값을 반환한다.

sys_sched_get_priority_min봉사루틴은 current가 실시간프로쎄스이면 1, 그렇지 않으면 0을 반환한다.

sys_sched_get_priority_max()봉사루틴은 current가 실시간프로쎄스면 가장 높은 우선순위인 99를, 그렇지 않으면 0을 반환한다.

- **sched_rr_get_interval()체계호출**

sched_rr_get_interval()체계호출은 pid변수로 지정한 실시간프로쎄스의 Round Robin시간정량을 사용자방식의 주소공간에 있는 구조체에 기록한다. pid가 0이면 이 체계호출은 현재프로쎄스의 시간정량을 기록한다.

해당 sched_rr_get_interval()봉사루틴은 마찬가지로 find_process_by_pid()를 호출하여 pid에 해당하는 프로쎄스서술자를 가져온다. 그리고 선택한 프로쎄스서술자의 nice마당에 들어있는 tick의 수를 초와 나노초(nanosecond)로 변환하여 이 값을 사용자방식에 있는 구조체로 복사한다.

제 5 절. 프로쎄스통신

이 절에서는 사용자방식프로쎄스가 서로 동기화하고 자료를 교환하는 방법을 설명한다. 프로쎄스는 프로쎄스호상간의 동기화와 통신을 위해서 핵심부에 의존해야만 한다.

앞의 《Linux파일잠그기》에서 살펴본것처럼 파일을 생성하고 이 파일에 열쇠를 걸고 해제하는 적절한 VFS체계호출을 하여 사용자방식프로쎄스사이의 동기화를 이룰수

있다. 이와 비슷한 방법으로 프로쎄스는 열쇠로 보호한 림시파일을 사용하여 자료를 공유할수 있지만 디스크파일체계에 접근해야 하므로 비용이 많이 든다. 이런 리유로 모든 Unix핵심부는 파일체계를 리용하지 않고 프로쎄스통신을 지원하는 체계호출을 포함한다. 더 나아가서 핵심부에 동기화를 요청하는 프로쎄스의 요구사항을 만족하기 위해 여러 래퍼함수를 적절한 서고에 추가하였다.

응용프로그람 개발자들의 다양한 요청으로 하여 다양한 통신기구가 필요하다. 다음은 Unix체계가 프로쎄스간 통신을 위해 제공하는 기본적인 기구이다.

- **관(PIPE)과 FIFO(이름붙은 관(named pipe))**

프로쎄스간에 생산자/소비자 관계의 호상작용을 실현하는데 가장 적당하다. 한 프로쎄스는 관에 자료를 넣고 다른 프로쎄스는 관에서 자료를 꺼낸다.

- **신호기(semaphore)**

이름에서 알수 있는것처럼 신호기에서 설명할 핵심부신호기의 사용자방식판본이다.

- **통보문(message)**

미리 정의된 통보문대기렬에 통보문을 쓰고 읽음으로써 통보문(작은 자료블로크)을 교환한다.

- **공유기억기령역(Shared memory region)**

공유된 기억기블로크를 리용하여 프로쎄스간정보를 교환할수 있게 한다. 많은 량의 자료를 공유해야 하는 경우 가장 효률적인 통신방식이다.

- **소케트(Socket)**

망을 통해 서로 다른 콤퓨터사이에 자료를 교환할수 있게 해준다. 소케트는 같은 주콤퓨터에 있는 프로쎄스간 통신도구로 리용할수도 있다. 례를 들어 XWindow체계는 소케트를 사용하여 의뢰기프로그람과 X봉사기사이에서 자료를 교환한다.

1. 관

관은 모든 Unix에서 제공하는 프로쎄스사이의 통신기구이다. 관은 프로쎄스사이에서 한 방향으로 흘러가는 자료흐름이다. 즉 한 프로쎄스가 관에 기록한 자료를 핵심부가 다른 프로쎄스로 전달하며 다른 프로쎄스는 해당 자료를 읽을수 있다.

Unix의 명령쉘에서 |연산자를 리용하여 관을 생성할수 있다. 례를 들어 다음 명령은 관으로 련결된 두 프로쎄스를 생성하도록 쉘에 명령한다.

```
$ ls | more
```

ls프로그람을 실행하는 첫번째 프로쎄스는 표준출력을 관으로 보내고 more프로그람을 실행하는 두번째 프로쎄스는 관에서 입력값을 읽는다.

다음과 같이 두 명령을 실행하여 같은 결과를 얻을수도 있다.

```
$ ls > temp
$ more < temp
```

첫번째 명령은 ls의 출력을 일반파일로 보낸다. 그 다음 두번째 명령은 more를 사용하여 같은 파일에서 입력값을 읽어들인다. 물론 림시파일대신 관을 사용하면 다음과 같은 우점이 있다.

- 쉘문장이 더 짧고 간단하다.
- 림시파일을 생성하고 삭제할 필요가 없다.

1) 관사용하기

관은 대응하는 영상이 탑재된 파일체계에 없는 열린 파일로 생각할수 있다. 프로쎄스는 pipe()체계호출을 사용하여 새로운 관을 생성한다. 이 체계호출은 파일서술자 두개를 반환한다. 프로쎄스는 이 서술자를 fork()체계호출을 사용하여 자식프로쎄스에 넘겨서 두 프로쎄스사이에 관을 공유할수 있다. 프로쎄스는 첫번째 파일서술자에 대해 read()체계호출을 하여 관에서 읽을수 있다. 이와 비슷하게 두번째 파일서술자에 대해 write()체계호출을 하여 관에 기록할수 있다.

POSIX는 단방향(반이중)관만을 정의하기때문에 pipe()체계호출이 파일서술자 두개를 반환하더라도 각 프로쎄스는 서술자를 사용하기 전에 다른 서술자를 닫아야 한다. 정방향 자료흐름이 필요하다면 프로쎄스는 pipe()체계호출을 두번 호출하여 서로 다른 관 두개를 리용해야 한다.

System V Release 4와 같은 Unix체계는 정방향(전이중)관을 지원한다. 정방향관에서는 두 서술자가 동시에 읽고 쓸수 있기때문에 정방향통로가 존재한다.

Linux는 또 다른 접근방법을 채택하였다. 매 관의 파일서술자는 여전히 단방향이지만 한 서술자를 사용하기 전에 다른 서술자를 닫을 필요가 없다.

앞에서 본 실례에서 ls | more문장을 해석하면 다음과 같은 동작을 실행한다.

1. pipe()체계호출을 한다. 여기서는 pipe()가 파일서술자 3(읽기통로)과 4(쓰기통로)를 반환한다고 가정한다.
2. fork()체계호출을 두번 한다.
3. close()체계호출을 두번 하여 파일서술자 3과 4를 해제한다.

ls프로그람을 실행하는 첫번째 자식프로쎄스는 다음 연산을 수행한다.

1. dup2(4,1)를 호출하여 파일서술자 4를 파일서술자 1로 복사한다. 이제부터 파일서술자 1은 관의 쓰기통로를 가리킨다.
2. close()체계호출을 두번 하여 파일서술자 3과 4를 해제한다.
3. execve()체계호출을 하여 ls프로그람을 실행한다. 이 프로그람은 출력을 파일서술자 1(표준출력)에 기록한다. 즉 이 프로그람은 관에 기록한다.

more프로그람을 실행하는 두번째 자식프로쎄스는 다음 연산을 수행한다.

1. dup2(3, 0)를 호출하여 파일서술자 3을 파일서술자 0으로 복사한다. 이제 파일서술자 0은 관의 읽기통로를 가리킨다.

2. close()체계호출을 두번 하여 파일서술자 3과 4를 해제한다.
3. execve()체계호출을 하여 more를 실행한다. 기본적으로 이 프로그람은 파일서술자 0(표준입력)에서 입력을 읽어들인다. 즉 관에서 읽어들인다.

이 실례에서는 두 프로세스가 관을 리용한다. 그러나 관실현에 따르면 여러 프로세스에서 한 관을 리용할수 있다. 물론 두 프로세스 이상이 동일한 관에서 읽거나 기록한다면 파일잠그기를 리용하거나 IPC신호기를 리용하여 관접근을 암시적으로 동기화해야 한다.

대부분의 Unix체계에서 pipe()체계호출외에도 popen()과 pclose()라는 두 래퍼함수를 제공한다. 이 함수들은 관을 리용할 때 발생하는 모든 일들을 처리한다. popen()함수를 리용하여 관을 생성하면 C서고에 포함된 고수준 입출력함수(fprintf(), fscanf() 등)로 관을 리용할수 있다.

Linux에서는 C서고에 popen()과 pclose()함수를 포함한다. popen()함수는 변수 두개를 받아들인다. 첫번째 변수인 filename은 실행파일의 경로이름이며 두번째 변수 type은 자료의 전송방향을 지정하는 문자렬이다. 이 함수는 FILE구조체의 지적자를 반환한다. popen()함수는 다음 연산을 수행한다.

1. pipe()체계호출을 사용하여 새로운 관을 생성한다.

2. 새로운 프로세스를 생성(fork)한 후 다음과 같은 연산을 수행한다.

a. type가 r이면 관의 쓰기통로와 관련된 파일서술자를 파일서술자 1(표준출력)로 복제한다. 이와 달리 type가 w이면 관의 읽기통로와 관련된 파일서술자를 파일서술자 0(표준입력)으로 복제한다.

b. pipe()가 반환한 파일서술자를 닫는다.

c. execve()체계호출을 하여 filename변수에 지적한 실행파일을 실행한다.

3. type가 r이면 관의 쓰기통로에 관련된 파일서술자를 닫는다. 이와 달리 type가 w이면 관의 읽기통로와 관련된 파일서술자를 닫는다.

4. 계속 열려있는 관에 대한 파일서술자를 가리키는 파일지적자 FILE의 주소를 반환한다.

popen()함수를 호출하면 부모와 자식프로세스가 관을 통해 정보를 교환할수 있다. 부모프로세스는 popen()함수가 반환한 FILE지적자를 리용하여 자료를 읽거나(type가 r일 경우) 쓸(type가 w일 경우)수 있다. 자식프로세스가 실행한 프로그람은 표준출력으로 자료를 쓰거나 표준입력에서 자료를 읽을수 있다.

popen()에서 반환한 파일지적자를 변수로 받는 pclose()함수는 wait4()체계호출을 하고 popen()이 생성한 프로세스가 완료할 때까지 기다린다.

2) 관자료구조

관의 자료구조를 살펴보려면 다시 체계호출수준에서 생각해야 한다. 관이 생성되면

프로쎄스는 read()와 write()와 같은 VFS체계호출을 사용하여 관에 접근할수 있다. 따라서 각 관에 대해서 핵심부는 i마디객체 하나와 파일객체 두개를 생성한다. 파일객체는 각각 읽기와 쓰기에 사용된다. 프로쎄스가 관에서 읽거나 쓰려면 적당한 파일서술자를 사용해야 한다. i마디객체가 관을 나타내면 i_pipe령역은 표 3-12에서 보여주는것처럼 관의 inode_info구조체를 가리킨다.

표 3-12. pipe_inode_info구조체

형	마당	설명
struct wait_queue *	Wait	관/FIFO대기렬
char *	Base	핵심부완충기 주소
unsigned int	Len	한 프로쎄스가 관에 쓰고 다른 프로쎄스가 아직 읽지 않는 바이트수
unsigned int	Start	핵심부완충기안에서의 읽기위치
unsigned int	readers	읽기프로쎄스를 위한 기발 또는 읽기프로쎄스의 수
unsigned int	writers	쓰기프로쎄스를 위한 기발 또는 쓰기프로쎄스수
unsigned int	waiting_readers	대기렬에서 잠들어있는 읽기프로쎄스수
unsigned int	waiting_writers	대기렬에서 잠들어있는 쓰기프로쎄스수
unsigned int	r_counter	FIFO를 읽을 때 사용하는 프로쎄스를 대기할 때 사용
unsigned int	w_counter	FIFO를 쓸 때 사용하는 프로쎄스를 대기할 때 사용

i마디 하나와 파일객체 두개외에도 매 관은 자신만의 관완충기 즉 관에 쓴 다음 아직 읽지 않은 자료가 있는 페지를 하나를 가진다. 이 페지들의 주소는 pipe_inode_info 구조체의 base마당에 저장된다. 구조체의 len마당에는 관완충기에 쓴 아직 읽지 않은 다음 바이트의 수가 저장된다. len마당에 저장되는 이 수를 《현재관크기》라고 부른다.

관완충기는 원형이며 읽기프로쎄스나 쓰기프로쎄스 모두 접근할수 있다. 그러므로 핵심부는 반드시 완충기에서 다음 두 편위의 현재 위치를 유지해야 한다.

- 다음에 읽을 바이트의 편위. 이 편위는 pipe_inode_info구조체의 state마당에 저장된다.
- 다음에 쓰기 위한 바이트의 편위. 이 편위는 start와 관크기(구조체의 len마당)

에서 구할수 있다.

핵심부는 관의 자료구조체에서 경쟁조건(race condition)을 피하기 위해 inode객체에 있는 i_sem신호기를 사용하여 관완충기에 대한 동시접근을 방지한다.

3) 특수파일체계 pipefs

관은 VFS객체의 집합으로 실현하며 대응하는 디스크영상이 없다. Linux 2.6핵심부에서는 이러한 동작을 처리하기 위해 VFS객체를 pipefs라는 특수파일체계에 실현하였다. pipefs파일체계는 체계등록부에 탑재위치가 없으며 사용자는 이 파일체계를 확인할수 없다. 그러나 pipefs로 인해 관은 VFS계층에 완전히 통합되였으며 핵심부는 사용자가 다시 인식할수 있는 이름붙은 관(named pipe)이나 FIFO와 같은 방법으로 다룰수 있게 되였다.(《FIFO》참고)

일반적으로 핵심부초기화시 실행하는 init_pipe_fs()함수는 pipefs파일체계를 등록한 후 탑재한다.(《뿌리파일체계 탑재》를 참고)

```
struct file_system_type pipe_fs_type;
root_fs_type.name = "pipefs";
root_fs_type_read_super = pipefs_read_super;
root_fs_type_fs_flags = FS_NOMOUNT;,
pipe_mnt = do_kern_mount ("pipefs", 0, “pipefs", NULL) ;
```

pipefs의 뿌리등록부를 나타내는 탑재된 파일체계 객체는 pipe_mnt변수에 저장된다.

4) 관의 생성과 제거

pipe()체계호출의 봉사루틴은 sys_pipe()함수인데 이 함수는 do_pipe()함수를 호출한다. do_pipe()는 새로운 관을 생성하기 위해 다음연산을 수행한다.

1. get_pipe_inode()함수를 호출한다. 이 함수는 pipefs파일체계에서 관에 대한 i마디객체를 할당하고 초기화한다. 이 함수는 다음동작을 실행한다.

   a. pipe_inode_info자료구조를 할당하고 그 주소를 inode의 i_pipe마당에 저장한다.

   b. 관완충기를 위한 페지를을 할당하고 그 시작주소를 pipe_mode_info구조체의 base마당에 저장한다.

   c. pipe_inode_info구조체의 start, len, waiting_readers, waiting_write마당을 0으로 초기화한다.

   d. pipe_inode_info구조체의 r_counter와 w_countcr 마당을 1로 초기화한다.

2. pipe_inode_info구조체의 readers와 writers마당을 1로 설정한다.

3. 관의 읽기통로에 대한 파일객체와 파일서술자를 할당한 다음 파일객체의 f_flag마당을 O_RDONLY로 설정한다. 그리고 f_op마당에 read_pipe_fops표의 주소를 저장

한다.

4. 관의 쓰기통로에 대한 파일객체와 파일서술자에 할당한 다음 파일객체의 flag마당을 O_WRONLY로 설정한다. 그리고 f_op마당에 write_pipe_fops표의 주소를 저장한다.

5. 등록부입구점객체를 할당하고 이것을 리용하여 i마디객체와 파일객체 두개를 련결한다.(《공통파일모형》을 참고) 그리고 새로운 i마디를 pipefs파일체계에 추가한다.

6. 파일서술자 두개를 사용자방식프로쎄스로 반환한다.

pipe()체계호출을 한 프로쎄스는 새로운 관에 읽고쓰기 위해 접근할수 있는 유일한 프로쎄스이다. 관에 읽기와 쓰기 프로쎄스가 있음을 나타내기 위해 pipe_inode_info자료구조체의 readers와 writers마당을 1로 초기화한다. 일반적으로 어떤 프로쎄스가 관의 파일객체를 열고있을 때에만 파일객체에 대응하는 마당이 1로 설정된다. 대응하는 파일객체를 해제하면 어떤 프로쎄스도 더는 접근하지 않기때문에 마당을 0으로 설정한다.

새로운 프로쎄스를 생성해서 자식프로쎄스를 생성하는것은 readers마당값과 writers마당값을 증가시키지 않으므로 1보다 커지지 않는다. 그러나 부모프로쎄스에서 계속 사용하는 모든 파일객체의 사용계수기는 증가한다.(《clone(), fork(), vfork()체계호출》를 참고) 그러므로 부모프로쎄스가 죽더라도 객체는 해제되지 않으며 자식프로쎄스에서 리용하기 위해 관은 열린 상태가 지속된다.

프로쎄스가 관을 가리키는 파일서술자에 대해 close()체계호출을 할 때마다 핵심부는 대응하는 파일객체에 fput()함수를 실행한다. 이 함수는 실행할 때마다 사용계수기를 감소시킨다. 계수기가 0이 되면 함수는 파일연산의 release메쏘드를 호출한다.

release메쏘드는 파일이 읽기통로와 쓰기통로중 어느 쪽에 관련되여있는가에 따라 pipe_read_release()나 pipe_write_release()중 하나로 실현한다.

이 두 함수는 pipe_release()를 호출하는데 이 함수는 pipe_inode_info구조체의 readers 또는 writers마당을 0으로 설정한다. 또한 이 함수는 readers와 writers마당이 둘 다 0이면 관완충기를 포함하는 페지틀을 해제한다. 그렇지 않다면 관상태의 변화를 알수 있도록 관의 대기렬에 잠들어있는 프로쎄스를 활성화한다.

5) 관에서 읽기

관에서 자료를 얻으려는 프로쎄스는 관의 읽기통로와 관련한 서술자를 파일서술자로 지정하여 read()체계호출을 실행한다. 《read()와 write()체계호출》에서 설명한것처럼 핵심부는 적절한 파일객체에 대응하는 파일연산표에서 찾은 read메쏘드를 호출한다. 관의 경우 read_pipe_fops표의 read메쏘드입구점은 pipe_read()함수를 가리킨다.

pipe_read()함수는 매우 복잡한데 POSIX 표준에서 관의 read연산에 대해 여러가지 요구사항을 명시하고있기때문이다.

표 3-13은 크기(관완충기안에 있는 아직 읽지 않은 바이트수)가 p인 관에서 n바이

트를 요청하는 read()체계호출의 예상동작을 보여준다.

표 3-13. 관에서 n바이트 읽기

<table>
<tr><td rowspan="3">관 크기 p</td><td colspan="3">하나이상의 쓰기프로세스</td><td rowspan="3">쓰기프로세스 없음</td></tr>
<tr><td colspan="2">차단된 읽기</td><td rowspan="2">차단되지 않은 읽기</td></tr>
<tr><td>잠들어있는 쓰기 프로세스 있음</td><td>잠들어있는 쓰기 프로세스 없음</td></tr>
<tr><td>P = 0</td><td rowspan="2">n바이트를 복사하고 n을 반환한다. 관완충기가 비여있으면 자료를 기다린다.</td><td>자료를 기다린 후 복사하고 관크기를 반환한다.</td><td>-EAGAIN을 반환한다.</td><td>0을 반환한다.</td></tr>
<tr><td>0< p <n</td><td colspan="3">p바이트를 복사하고 p바이트를 반환한다. 관에는 0바이트가 남아있다.</td></tr>
<tr><td>p >= n</td><td colspan="4">n바이트를 복사하고 n바이트를 반환한다. 관완충기에서는 p-n바이트가 남아있다.</td></tr>
</table>

체계호출은 다음 두 경우에 현재프로세스를 차단한다.

- system호출을 호출했을 때 관완충기가 비여있다.
- 요청한 만큼의 바이트가 관완충기에 들어있지 않으며 이전호출에서 쓰기프로세스가 완충기에 빈 공간이 생길 때까지 잠든 상태로 들어갔다.

차단되지 않는 읽기연산도 있다. 이 경우 모든 가능한 바이트(리용가능한 바이트가 없을수도 있다.)를 사용자주소공간으로 복사하고 즉시 완료한다.

read()체계호출은 관이 비여있고 관의 쓰기통로와 관련한 파일객체를 리용중인 프로세스가 없을 때에만 0을 반환한다.

함수는 다음과 같은 연산을 수행한다.

1. inode의 i_sem신호기를 획득한다.

2. pipe_inode_info구조체의 len마당에 있는 관의 크기가 0인가를 검사한다. 관크기가 0이면 함수가 반드시 즉시 되돌이해야 하는지 또는 다른 프로세스가 관에 자료를 기록하는 동안 기다리기 위해 차단되여야 하는가를 결정한다.(표 3-13 참고) 입출력연산의 류형(차단이나 비차단)은 파일객체의 f_flags마당에 O_NONBLOCK기발을 리용하여 지정할수 있다. 만약 현재프로세스가 차단되여야 한다면 함수는 다음동작을 수행한다.

 a. pipe_inode_info구조체의 waiting_readers마당에 1을 더한다.

 b. 관의 대기렬(pipe_inode_info구조체의 wait마당)에 current값을 더한다.

 c. i마디 신호기를 해제한다.

 d. 프로세스상태를 TASK_INTERRUPTIBLE로 설정하고 schedule()함수를

호출한다.

e. 깨여나면 대기렬에서 current값을 제거하고 i_sem신호기를 다시 얻은 후 waiting_readers마당을 감소시키고 단계 2로 간다.

3. 관완충기에서 요청한 바이트수(또는 완충기크기가 너무 작다면 리용가능한 바이트수)를 사용자주소공간에 복사한다.

4. pipe_inode_info구조체의 start와 len마당을 갱신한다.

5. wake_up_interruptible함수를 호출하여 관의 대기렬에 잠들어있는 모든 프로쎄스를 깨운다.

6. 요청한만큼의 비이트를 복사하지 못했으며 현재 잠든(waiting_writers마당이 0보다 큰) 쓰기프로쎄스가 하나이상 있고 읽기연산이 차단이면 다시 단계 2로 되돌아간다.

7. i마디의 i_sem신호기를 해제한다.

8. 복사한 바이트수를 사용자주소공간으로 반환한다.

6) 관에 쓰기

관에 자료를 넣으려는 프로쎄스는 관의 쓰기통로와 관련한 파일서술자를 지정하여 write()함수를 호출한다. 핵심부는 파일객체의 쓰기메쏘드를 호출하여 이 동작을 수행한다. write_pipe_fops 표의 대응하는 입구점은 pipe_write() 함수를 가리킨다.

표 3-13에서는 완충기안에 u만큼 사용하지 않는 바이트를 포함한 관에 n바이트를 쓰려고 요청하는 write()체계호출에 대해 POSIX표준에서 지정하는 동작을 보여준다. 특히 POSIX표준에서는 작은 수의 바이트에 대한 쓰기연산은 반드시 한번에 실행해야 한다고 명시하고있다. 좀 더 자세히 말하자면 프로쎄스 두개 이상이 동시에 관에 쓰려고 할 때 크기가 4096B(관완충기 크기)이하인 쓰기연산은 다른 프로쎄스가 동일한 관에 대해 쓰기연산을 실행하는것과 뒤섞이지 않고 완료해야 한다. 그러나 4096B이상인 쓰기연산은 한번에 실행되지 못할수도 있으며 호출하는 프로쎄스를 잠든 상태로 만들수도 있다.

표 3-14. 관에 n바이트 쓰기

리용가능한 완충기공간 u	최소한 하나이상의 읽기프로쎄스		읽기프로쎄스 없음
	차단되는 쓰기	차단되지 않는 쓰기	
u< n <= 4096	n-u바이트가 여유있을 때까지 대기한 후 n바이트를 복사하고 n을 반환한다.	-EAGAIN을 반환한다.	SIGPIPE신호를 낸 후 EPIPE를 반환한다.
n > 4096	n바이트(필요하다면 기다림)를 복사하고 n을 반환한다.	u>0이면 u바이트 를 복사한 후 u바이트를 반환한다. 그렇지 않으면 EAGAIN을 반환한다.	

u >= n	n바이트를 복사하고 n바이트를 반환한다.

관에 대한 쓰기연산은 관에 읽기프로세스가 없으면(즉 관의 inode객체의 readers 마당이 0값인 경우) 반드시 실패한다. 이 경우 핵심부는 쓰기프로세스에 SIGPIPE신호를 보내고 -EPIPE오유코드와 함께 write()체계호출을 완료한다. 이 오유코드는 관깨짐(Broken pipe)통보문이다.

pipe_write()함수는 다음 동작을 수행한다.

1. i마디의 i_sem신호기를 얻는다.

2. 관에 읽기프로세스가 최소한 하나이상 있는가를 검사한다. 만약 없다면 현재프로세스에 SIGPIPE신호를 보내고 i마디신호기를 해제한 후 -EPIPE오유코드를 반환한다.

3. 기록할 바이트수가 관의 완충기크기보다 작은가를 검사한다.

   a. 만약 완충기크기보다 작다면 쓰기연산은 원자성을 만족해야 한다 그러므로 완충기크기가 기록하려는 모든 바이트를 저장할수 있을만큼 충분한가를 검사한다.

   b. 기록할 바이트수가 완충기크기보다 크다면 여유공간이 약간만 있더라도 연산을 바로 시작한다. 그러므로 함수는 여유바이트가 최소한 1B인가를 검사한다.

4. 만약 완충기에 충분한 공간이 없고 쓰기연산이 비차단이라면 i마디신호기를 해제하고 -EAGAIN오유코드를 반환한다.

5. 완충기에 충분한 공간이 없고 쓰기연산이 차단이라면 다음 동작을 수행한다.

   a. pipe_inode_info구조체의 waiting_writer마당에 1을 더한다.

   b. 관(pipe_inode_info 구조체의 wait 마당)의 대기렬에 current를 추가한다.

   c. i마디 신호기를 해제한다.

   d. 프로세스상태를 TASK_INTERRUPTIBLE로 설정하고 schedule()를 호출한다.

   e. 프로세스가 깨여나면 대기렬에서 current를 제거하고 i마디신호기를 다시 획득하고 waiting_writers마당값을 감소시킨다. 그리고 단계 5로 되돌아간다.

6. 이제 관완충기는 요청한 바이트수(쓰기연산에 원자성이 있어야 하는 경우)를 충분히 확보했거나 최소한 1바이트를 확보하였다. 이 쓰기프로세스가 i마디신호기를 소유하고있기때문에 다른 쓰기프로세스가 여유공간을 확보할수 없다. 요청한 바이트수(또는 관 크기가 너무 작다면 여유 바이트수)를 사용자주소공간에서 관완충기로 복사한다.

7. 관의 대기렬에 있는 잠들어있는 모든 프로세스를 깨운다.

8. 만약 쓰기연산이 차단이고 관완충기에 요청한 모든 바이트수를 기록하지 못하였다면 5단계로 되돌아간다. 이러한 경우는 쓰기연산에 원자성이 없을 경우에만 일어난다. 현재프로세스는 관완충기에 한 바이트라도 여유공간이 생길 때까지 차단된 상태로 남아있게 된다.

9. i마디신호기를 해제한다.

10. 관의 완충기에 기록한 바이트수를 반환한다.

2. FIFO

관은 단순하고 유연하며 효률적인 통신기구이다. 그렇지만 이미 존재하는 관을 열수 없다는 큰 부족점이 있다. 이 부족점때문에 공동조상프로쎄스가 생성한 관이 없다면 임의의 두 프로쎄스가 동일한 관을 공유할수 없다.

이 부족점은 많은 응용프로그람에 영향을 미친다. 례를 들어 자료기지엔진봉사기에서 봉사기가 질문을 요청하는 의뢰기프로쎄스를 련속적으로 받고 자료기지검색결과를 의뢰기에 되돌려주는 경우를 생각해보자.

봉사기와 각 의뢰기의 호상작용은 관을 리용하여 처리할수 있다. 그러나 사용자가 명령쉘을 리용하여 자료기지에 질문하기때문에 의뢰기프로쎄스는 사용자의 명령쉘에서 생성된다. 그러므로 봉사기와 의뢰기프로쎄스는 관을 쉽게 공유할수 없다.

Unix체계는 이러한 제한을 해결하기 위해 이름붙은 관(named pipe) 또는 FIFO라는 특별한 파일류형을 도입하였다. FIFO는 관과 거의 비슷하다. FIFO는 파일체계에서 디스크블로크를 소유하지 않으며 열린 FIFO는 프로쎄스 두개이상이 자료를 교환하기 위해 핵심부완충기를 림시 저장공간으로 사용한다.

디스크 i마디(inode)로 하여 FIFO파일명은 체계의 등록부나무에 포함되여있으므로 어떠한 프로쎄스도 FIFO에 접근할수 있다. 다시 자료기지실례를 보자. 자료기지봉사기와 의뢰기는 관대신 FIFO를 리용하여 서로간의 통신을 간단히 시작할수 있다. 시작할때 봉사기는 의뢰기프로그람의 요청을 처리하기 위한 FIFO를 생성한다. 각각의 의뢰기프로그람은 봉사기와 련결이 성립되기전에 봉사기프로그람이 의뢰기프로그람의 질문결과를 기록할수 있는 다른 FIFO를 생성하며 생성한 FIFO이름을 첫 요청에 포함한다.

Linux 2.6핵심부에서 FIFO와 관은 거의 동일하며 같은 pipe_inode_info구조체를 리용한다. 이러한 리유때문에 FIFO의 읽기와 쓰기 파일연산은 앞서 《관에서 읽기》와 《관에 쓰기》에서 언급한 동일한 pipe_read()와 pipe_write()함수로 실현되였다.

하지만 여기에는 두가지 큰 차이점이 있다.

- FIFO i마디는 pipefs라는 특별한 파일체계가 아닌 체계등록부나무에 있다.
- FIFO는 쌍방향통신통로다. 즉 한 FIFO를 읽기/쓰기방식으로 열수 있다.

이러한 내용을 좀 더 자세히 설명하기 위해 다음 부분에서는 FIFO를 생성하고 여는 방법을 설명한다.

프로쎄스는 mknod()체계호출을 하여 FIFO를 생성한다. FIFO를 생성할 때 새로운 FIFO경로이름과 새로운 파일허가비트마스크와 S_IFIFO(0x1000)값을 론리연산자 OR로 련결한 값을 변수로 전달한다. POSIX에서는 FIFO를 생성하기 위한 mkfifo()라는 특별한 함수를 도입하였다. System V Release 4와 같이 Linux에서는 이 함수를

mknod()를 호출하는 C서고함수로 실현하였다.

FIFO가 생성되면 open(), read(), write(), close()체계호출을 리용하여 FIFO에 접근할수 있지만 FIFO i마디와 파일연산이 변형되였으며 FIFO가 저장된 파일체계에 의존적이지 않기때문에 VFS는 특수한 방법으로 FIFO를 다룬다.

POSIX 표준은 FIFO에서 open()체계호출의 동작방식을 명시하고있다. 이러한 동작은 기본적으로 요청한 류형, 입출력연산의 종류(차단 또는 비차단), FIFO에 접근하는 다른 프로쎄스의 존재여부에 따라 다르다.

프로쎄스는 읽기나 쓰기용으로 또는 읽기/쓰기용으로 FIFO를 열수 있다. 파일연산은 이러한 3가지 경우에 따라 다른 메쏘드로 설정된다.

프로쎄스가 FIFO를 열면 VFS는 장치파일에서 다룬것과 동일한 연산을 수행한다. 열린 FIFO에 대응하는 i마디객체는 파일체계에 의존적인 read_inode초블로크메쏘드를 통해 초기화된다. 이 read_inode메쏘드는 언제나 디스크에 있는 i마디가 특수한 파일을 나타내는가를 검사하며 필요하면 init_special_inode()함수를 호출한다. 그 다음 이 함수는 i마디객체의 i_fop마당을 def_fifo_fops표의 주소로 설정한다. 이후에 핵심부는 파일객체의 파일연산표를 def_fifo_fops로 설정하고 fifo_open()으로 실현한 open메쏘드를 실행한다.

fifo_open()함수는 FIFO에 있는 자료구조를 초기화한다. 이 함수는 다음 연산을 수행한다.

1. i_sem inode신호기를 획득한다.

2. i마디객체의 i_pipe마당을 검사한다. 만약 이 마당이 NULL이면 새로운 pipe_inode_info구조체를 앞서 본 《관의 생성과 소멸》의 1단계와 동일하게 할당하고 초기화한다.

3. open()체계호출의 변수에서 지정한 접근방식에 따라 파일객체의 f_op마당을 적절한 파일연산표의 주소로 초기화한다.(표 3-15 참고)

표 3-15. FIFO의 파일연산

접근류형	파일연산	읽기메쏘드	쓰기메쏘드
읽기전용	read_fifo_fops	pipe_read()	bad_pipe_w()
쓰기전용	write_fifo_fops	bad_pipe_r()	pipe_write()
읽고쓰기	rdwr_fifo_fops	pipe_read()	pipe_write()

4. 만약 접근방식이 읽기전용이나 읽기쓰기로 되여있다면 pipe_inode_info구조체의 readers와 r_counter마당에 하나씩 더한다. 만약 접근방식이 읽기전용이고 어떠한 읽기프로쎄스도 없다면 대기렬에서 잠들어있는 쓰기프로쎄스를 깨운다.

5. 만약 접근방식이 쓰기전용이거나 읽기쓰기로 되여있다면 pipe_inode_info구조체의 wnter와 w_counter마당에 하나씩 더한다. 만약 접근방식이 쓰기전용이고 쓰기프

교육성 교육정보쎈터

로프로쎄스가 없다면 대기렬에서 잠들어있는 읽기프로쎄스를 깨운다.

6. 읽기프로쎄스와 쓰기프로쎄스가 모두 없다면 함수를 차단할것인지, 오유코드(표 3-16)를 반환하여 완료할것인지를 결정한다.

표 3-16. fifo_open()함수의 동작

접근 류형	차단하기	비차단하기
읽기전용, 쓰기프로쎄스	성공적으로 반환	성공적으로 반환
읽기전용, 쓰기프로쎄스 없음	쓰기프로쎄스를 기다림	성공적으로 반환
쓰기전용, 읽기 프로쎄스	성공적으로 반환	성공적으로 반환
쓰기전용, 읽기프로쎄스 없음	읽기프로쎄스를 기다림	-ENXIO반환
읽기/쓰기	성공적으로 반환	성공적으로 반환

7. i마디신호기를 해제한 후 완료하고 0(성공)을 반환한다 .

FIFO의 3가지 파일연산표는 read와 write메쏘드의 실현방법에서 차이가 난다. 접근류형이 읽기연산을 허용한 경우 읽기메쏘드는 pipe_read()함수로 실현하고 그렇지 않으면 오유코드를 반환하는 bad_pipe_r()로 실현한다. 이와 비슷하게 접근류형이 쓰기방식를 허용한다면 쓰기메쏘드는 pipe_write()함수로 실현하고 그렇지 않으면 오유코드를 반환하는 bad_pipe_w()로 실현한다.

3. System V IPC

IPC는 프로쎄스간통신(Interprocess Communication)의 략자로서 사용자방식프로쎄스가 다음과 같은 일을 할수 있게 하는 체계호출집합을 나타낸다.

- 신호기를 리용하여 다른 프로쎄스와 동기화하기
- 다른 프로쎄스로 통보문을 보내거나 다른 프로쎄스로부터 통보문받기
- 다른 프로쎄스와 기억기령역 공유하기

System V IPC는 Linux를 포함한 대부분의 Unix체계에서 사용한다.

IPC자료구조는 프로쎄스가 IPC자원(신호기, 통보문대기렬이나 공유기억기령역 등)을 요청할 때마다 동적으로 생성된다. 매 IPC자원은 지속적으로서(persistent) 프로쎄스가 명시적으로 제거하지 않으면 체계를 끌낼 때까지 계속 기억기에 남아있다. IPC자원을 생성한 프로쎄스와 조상프로쎄스를 공유하는가와는 관계없이 모든 프로쎄스가 이 자원을 리용할수 있다.

프로쎄스가 같은 류형의 IPC자원을 여러개 요청할수 있으므로 매개의 새로운 자원은 32bit IPC단어로 구별한다. 이 단어는 체계등록부나무의 파일경로이름과 비슷하다. 또한 각 IPC자원은 IPC식별자를 가진다. 이것은 파일인 경우의 파일서술자와 비슷하다. IPC식별자는 핵심부가 할당하며 체계안에서 유일하고 IPC단어는 프로그람작성자가 임

의로 선택할수 있다.

두개이상의 프로쎄스가 IPC자원을 리용하여 통신하려면 자원의 IPC식별자를 참조한다.

1) IPC자원리용

자원은 새로운 자원이 신호기, 통보문대기렬이나 공유기억기령역인가에 따라 semget(), msgget()나 shmget()함수를 호출함으로써 생성된다. 이러한 함수들의 기본목적은 IPC단어(첫번째 변수)에 대응하는 IPC식별자를 얻는것이다. 프로쎄스는 자원에 접근하기 위해 이 식별자를 리용한다. 만약 IPC단어에 대응하는 IPC자원이 없다면 새로운 자원이 생성된다. 모든 일이 제대로 되였다면 함수는 정수인 IPC식별자를 반환하고 그렇지 않다면 표 3-17에서 보여주는 오유코드를 반환한다.

표 3-17. IPC식별자를 얻는 과정에서 반환하는 오유코드

오유코드	설 명
EACCESS	프로쎄스가 적절한 접근권한이 없다.
EEXIST	프로쎄스가 이미 존재하는 열쇠를 리용하여 IPC를 생성하려고 시도한다.
EIDRM	삭제될 자원으로 표시된다.
ENOENT	요청한 열쇠를 소유한 IPC자원이 없으며 프로쎄스는 생성을 요청하지 않는다.
ENOMEM	추가적인 IPC자원을 저장할 공간이 없다.
ENOSPC	최대IPC자원수제한을 초과한다.

독립적인 두 프로쎄스가 공통IPC자원을 공유하기를 원한다고 가정해보자. 이것은 다음 두가지 방법으로 수행할수 있다.

- 프로쎄스는 미리 정의되여있는 고정된 IPC단어를 사용한다. 이것은 매우 간단한 경우이며 많은 프로쎄스로 실현한 복잡한 응용프로그람에서도 잘 동작한다. 그러나 관계없는 다른 프로그람이 동일한 IPC단어를 결정하는 경우가 있다. 이러한 경우라도 IPC함수는 성공적으로 호출되지만 잘못된 자원의 IPC식별자를 반환할수 있다.
- 한 프로쎄스가 IPC단어로 IPC_PRIVATE를 지정하여 semget(), msget()나 shmget()함수를 호출한다. 그러면 새로운 IPC자원이 할당되며 프로쎄스는 응용프로그람의 다른 프로쎄스에 자신의 IPC를 전하거나 다른 프로쎄스를 직접 생성할수 있다. 이러한 방법은 다른 응용프로그람에서 우연히 IPC자원을 리용하는것을 막아준다. semget(), msgget()와 shmget()함수의 마지막 변수는 두 기발을 포함

할수 있다.

IPC_CREAT는 이미 IPC자원이 존재하지 않으면 반드시 생성해야 한다. 이와 반대로 IPC_EXCL은 이미 자원이 존재하고 IPC_CREAT기발이 설정되여있으면 함수가 반드시 실패함을 나타낸다.

심지어 프로쎄스가 IPC_CREAT와 IPC_EXCL기발을 리용하더라도 다른 프로쎄스가 IPC식별자를 리용하여 자원을 참조할수 있기때문에 IPC자원에 배타적접근을 보장할수 있는 방법은 없다.

핵심부는 이렇게 잘못된 자원을 참조하는 위험을 최소화하기 위해 IPC식별자가 여유있게 되더라도 재사용하지 않는다. 대신 자원에 할당하는 IPC식별자는 이전에 동일한 류형의 자원에 할당된 식별자보다 항상 크다.(32bit IPC식별자가 초파(overflow)한 경우는 례외이다.) 매개의 IPC식별자는 자원류형과 관련된 슬로트사용순서번호(slot usage sequence number)와 할당된 자원에 대한 임의의 슬로트색인값 그리고 할당가능한 자원의 최대수보다 더 크며 핵심부가 선택한 임의의 수조합으로 계산된다. s가 슬로트사용순서번호를 나타내고 M은 할당가능한 자원의 최대수를 나타내며 i가 슬로트색인값을 나타낸다면(0<=i<M) 각 IPC자원의 ID는 다음과 같이 계산된다.

IPC 식별자 = s * M + I

Linux 2.6에서는 M의 값은 32768(IPCMIN 마크로)로 설정되여있다. 슬로트사용순서번호 s는 0으로 초기화되며 각 자원을 할당할 때마다 하나씩 증가한다. s가 IPC자원의 류형에 따라 미리 정의된 값(threshold)에 도달하게 되면 다시 0부터 시작하게 된다.

IPC자원의 모든 류형(신호기, 통보문대기렬 그리고 공유기억기령역 등)은 ipc_ids 자료구조를 가지고있으며 이 자료구조의 마당값을 표 3-18에 주었다.

표 3-18. ipc_ids자료구조마당

형	마 당	설 명
Int	size	IPC자원의 현재 최대값
Int	in_use	할당된 IPC자원의 수
Int	max_id	사용중인 슬로트색인의 최대값
unsigned short	seq	다음 할당에 대한 슬로트사용순서번호
unsigned short	seq_max	최대슬로트사용순서번호
struct semaphore	sem	ips_ids자료구조를 보호하는 신호기
spinlock_t	ary	IPC자원서술자를 보호하는 스핀잠그기
struct ipd_id *	entries	IPC자원서술자의 배렬

size마당은 주어진 류형에서 할당가능한 IPC자원의 최대수를 저장한다. 체계관리자

는 각각 /proc/sys/kernel/sem, /proc/sys/kernel/msgmni 그리고 /proc/sys/kernel/shmmni 의 값을 조정함으로써 매 자원의 size마당값을 증가시킬수 있다.

entries마당값은 할당가능한 모든 자원(size마당 또한 배렬의 크기이다.)마다 하나씩 대응하는 kern_ipc_perm자료구조의 배렬지적자를 가리킨다. IPC자원과 대응하는 kem_ipc_perm자료구조의 각 마당값을 표 3-18에 보여준다. uid, gid, cuid와 cgid마당은 각각 자원을 생성한 생성자의 사용자와 그룹식별자를 저장하고 현재자원의 소유자에 대한 사용자와 그룹식별자를 저장한다. mode비트마스크는 기발 6개를 포함하며 매 자원의 소유자, 자원의 그룹, 그 외의 사용자들에 대한 읽기와 쓰기 접근권한을 저장한다. IPC접근권한은 《접근 권한과 파일방식》에서 설명한 파일접근권한과 비슷하다. 그러나 IPC접근권한에서는 실행허가기발을 사용하지 않는다.

kern_ipc_perm자료구조는 key마당(자원에 대응하는 IPC단어를 포함)과 seq마당(자원에 대한 IPC식별자를 계산하는데 리용하는 슬로트사용순서번호, 앞에 있는 수식의 s에 해당)도 포함하고있다.

표 3-19. kern_ipc_perm구조체의 마당

형	마 당	설명
int	key	IPC열쇠
unsigned int	uid	소유자의 사용자 ID
unsigned int	gid	소유자의 그룹 ID
unsigned int	cuid	생성자의 사용자 ID
unsigned int	cgid	생성자의 그룹 ID
unsigned short	mode	허가비트 마스크
unsigned long	seq	슬로트사용순서번호

semctl(), msgctl()과 shmctl()함수를 사용하여 IPC자원을 처리할수 있다. IPC_SET명령은 프로쎄스가 ipc_perm자료구조에 있는 허가비트마스크와 소유자의 사용자와 그룹식별자를 변경할수 있게 한다. IPC_STAT와 IPC_INFO명령은 자원에 관련한 일부 정보를 얻는다. 마지막으로 IPC_RMID명령은 IPC자원을 해제한다. IPC자원의 류형에 따라 다른 특수한 명령을 리용할수 있다.

IPC자원이 생성되면 프로쎄스는 일부 특수한 함수를 리용하여 자원을 사용할수 있다. 프로쎄스는 semop()함수를 호출함으로써 IPC신호기를 얻거나 해제할수 있다. 프로쎄스가 IPC통보문을 보내거나 받으려고 한다면 msgsnd()함수와 msgrcv()함수를 리용한다. 마지막으로 프로쎄스는 shmat()와 shmdt()함수를 리용하여 IPC공유기억기를 자신의 주소공간에 붙이거나 뗀다.

2) ipc()체계호출

모든 IPC함수는 적절한 Linux체계호출을 리용하여 실현해야 한다. 실제로 80x86

방식에서는 ipc()라는 단 하나의 IPC체계호출만이 있을뿐이다. 프로쎄스가 IPC함수 례를 들어 msgget()함수를 호출하면 사실은 C서고에 있는 래퍼함수를 호출하는것이다. 이 함수는 ipc()체계함수를 호출하는데 msgget()의 변수와 MSGGET명령을 전달한다. sys_ipc()봉사루틴은 명령을 검사하고 요청된 봉사를 실현하는 핵심부함수를 호출한다.

ipc() 《다중화장치(multiplexer)》체계호출은 동적모듈에 IPC코드를 포함하는 이전 Linux판본에서 유래되였다. 제거가능한 핵심부구성요소를 위해 system_call표에 여러 체계호출을 예약하는 일은 비합리적이므로 핵심부설계자는 다중화장치방식을 채택하였다.

지금은 System V IPC를 더는 동적모듈로 콤파일할수 없으며 단일 IPC체계호출을 사용할 리유가 없다. 사실 Linux는 HP Alpha방식과 Intel IA-64에서는 각 IPC함수에 대해 체계호출을 하나씩 제공한다.

3) IPC신호기

IPC신호기는 핵심부신호기와 대단히 류사하다. 이 신호기들은 여러 프로쎄스가 공유하는 자료구조에 대한 통제된 접근방법을 제공하기 위해 사용하는 계수기이다.

신호기값은 보호된 자원이 리용가능하면 정수이며 보호된 자원이 현재 리용불가능하면 0이다. 자원에 접근하려는 프로쎄스는 신호기값을 감소시킨다. 그러나 신호기값이 0이면 핵심부는 신호기값이 정수가 될 때까지 프로쎄스를 차단한다. 프로쎄스가 보호된 자원을 해제하면 신호기값을 감소시키고 이렇게 함으로써 신호기에 대기중인 다른 프로쎄스가 깨여나게 되는것이다.

실제로 IPC 신호기는 다음 두가지 리유로 인해 핵심부신호기보다 좀더 다루기 어렵다.

· 각 IPC신호기는 핵심부신호기처럼 단일값이 아니라 신호기값 하나이상을 가지는 집합이다. 즉 한 IPC자원이 여러 독립적인 공유자료구조를 보호할수 있다는 의미이다. 각 신호기에 있는 신호기값의 수는 자원을 할당할 때 semget()함수의 변수로 명시해야 한다. 이제부터 IPC신호기 내부의 계수기를 기본신호기(primitive semaphore)라고 한다. 두가지 제한값이 있는데 하나는 IPC신호기 자원의 수(기본적으로 128)이고 다른 하나는 단일 IPC신호기 자원내의 기본신호기 수(기본적으로 250)이다. 그러나 체계관리자는 《/proc/sys/kernel/sem》파일에 다른 수자를 기록함으로써 이 제한값을 쉽게 변경할수 있다.

· System V IPC신호기에는 프로쎄스가 신호기에 내린 이전연산을 취소하지 못하고 죽는 상태에 대한 실패-안전(fail-safe)기구가 있다. 프로쎄스가 이 기구를 사용하는 경우 이런 연산을 취소가능(undoable)신호기연산이라고 한다. 프로쎄스가 죽을 때 프로쎄스의 모든 IPC신호기는 프로쎄스가 연산을 시작하지 않았을 경우 가졌을 값으로 되돌아갈수 있다. 이렇게 함으로써 신호기연산을 취소하는데 실패함으로 인해 같은 신호기를 사용하는 다른 프로쎄스가 차단된 상태로 계속 남아있는것을 방지한다.

먼저 IPC신호기로 보호된 자원 하나이상에 접근하려는 프로쎄스동작의 전형적인 단계를 살펴보자.

1. IPC신호기식별자를 얻기 위해 semget()래퍼함수를 호출한다. 변수로는 공유자원을 보호하는 IPC신호기의 IPC단어를 전달한다. 프로쎄스가 새로운 IPC신호기를 생

성하려면 IPC_CRATE 또는 IPC_PRIVATE기발과 필요한 기본신호기의 수도 지정한다.(앞서 본 《IPC자원 리용하기》를 참고)

2. semop()래퍼함수를 호출하여 모든 기본신호기값을 검사하고 감소시킨다. 모든 검사가 성공하면 신호기값을 줄이고 완료한다. 그러면 프로쎄스는 보호된 자원에 접근할수 있게 된다. 이미 사용중인 신호기가 있다면 프로쎄스는 다른 프로쎄스가 자원을 해제할 때까지 보류된다. 함수는 IPC신호기식별자와 기본신호기에서 자동으로 실행되기 위한 지정한 연산을 나타내는 정수의 배렬과 이러한 연산의 수를 변수로 받는다. 프로쎄스는 선택항목으로 SEM_UNDO기발을 지정할수 있는데 이 기발은 프로쎄스가 기본신호기를 해제하지 않고 완료한 경우 핵심부가 연산을 되돌리도록 지시한다.

3. 보호된 자원을 풀어줄 때 semop()함수를 다시 호출하여 관련된 모든 기본신호기를 한번에 증가시킨다.

4. 선택항목으로 IPC_RMID명령을 지정하여 semctl()래퍼함수를 호출하여 체계에서 IPC신호기를 제거한다.

이제 핵심부에서 IPC신호기를 실현하는 방법을 론의할수 있다. 관련된 자료구조는 그림 3-15와 같다. sem_ids변수는 IPC신호기자원류형인 ipc_ids자료구조를 저장한다. ipc_ids자료구조의 entries마당은 sem_array자료구조를 가리키는 지적자의 배렬이며 각 항목은 IPC신호기자원 하나씩을 가리킨다.

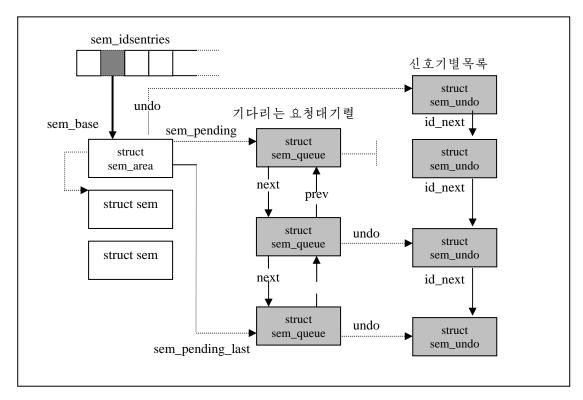



그림 3-15. IPC 신호기자료구조

형식적으로 배렬은 kem_ipc_perm자료구조의 지적자를 저장하지만 사실 매개의 자

료구조는 sem_array자료구조의 첫번째 마당이다. sem_array자료구조의 모든 마당은 표 3-20에 있다.

표 3-20. sem_array자료구조의 마당

형	마 당	설 명
struct kern_ipc_perm	sem_perm	kern_ipc_perm자료구조
long	sem_otime	마지막 semop()의 시간형
long	sem_ctime	마지막 변경에 대한 시간형
struct sem *	sem_base	처음 sem구조체를 가리키는 지적자
struct sem_queue *	sem_pending	대기중인 연산
struct sem_queue **	sem_pending_last	마지막 대기중인 연산
struct sem_undo *	undo	취소(undo)요청
unsigned short	sem_nsems	배렬에 있는 신호기수

sem_base마당은 struct sem자료구조의 배렬을 가리킨다. 배렬의 각 자료구조는 IPC신호기 하나를 나타낸다. 이 자료구조는 다음 두 마당을 포함한다.

semval

신호기의 계수기 값

sempid

신호기에 마지막으로 접근한 프로쎄스의 PID. 이 PID값은 프로쎄스가 semctl()래퍼함수를 사용하여 질문할수 있다.

4) 취소가능한 신호기연산

프로쎄스가 갑자기 멈추면 프로쎄스는 시작한 연산을 취소할수 없다.(례를 들면 예약한 신호기해제) 따라서 프로쎄스를 《취소가능》으로 선언하여 핵심부가 신호기를 이전의 일관성있는 상태로 되돌리고 다른 프로쎄스가 작업을 계속 진행하게 한다. 프로쎄스는 SEM_UNDO기발을 지정해서 sem_op()함수를 호출하여 취소가능한 연산을 요청할수 있다.

sem_undo자료구조에는 정해진 어떤 IPC신호기자원에 대해 프로쎄스가 수행한 취소가능한 동작연산을 핵심부가 되돌리는것을 돕기 위한 정보가 저장되여있다. 이것은 핵심적으로 신호기의 IPC식별자와 프로쎄스가 수행한 취소가능한 연산의 수행결과로 발생한 기본신호기의 변경값을 나타내는 정수의 배렬을 포함하고있다.

sem_undo요소를 리용하는 방법을 간단히 설명하면 다음과 같다. 기본신호기 4개를 포함하고 IPC신호기 자원을 리용하는 프로쎄스가 있으며 이 프로쎄스는 처음 계수기를 하나 증가시키고 두번째 계수기를 2씩 감소시키는 semop()함수를 호출한다고 가정하자.이 프로쎄스가 SEM_UNDO기발을 지정하면 sem_undo자료구조의 첫번째 배렬요

소에 있는 정수는 하나 감소하고 두번째 요소에 있는 정수는 둘 증가할것이며 다른 두 정수는 바뀌지 않을것이다.

동일한 프로쎄스에서 수행하는 IPC신호기에 대한 취소가능한 연산은 sem_undo구조체의 정수값을 적절하게 바꾼다. 프로쎄스가 완료되면 배렬에 있는 0이 아닌 값은 대응하는 기본신호기에 있는 불완전한 연산과 대응한다. 핵심부는 대응하는 신호기의 계수기에 0이 아닌 값을 더함으로써 이 연산들을 되돌린다. 다시 말하면 중단된 처리기가 변경한 값은 취소하며 다른 프로쎄스가 변경한 값은 신호기의 상태에 계속 반영한다.

각 프로쎄스에 대해서 핵심부는 취소가능한 연산으로 처리하는 모든 신호기자원을 추적하여 프로쎄스가 예상하지 못한 완료를 하게 되면 다시 취소할수 있게 한다. 그리고 매 신호기에 대해 핵심부는 모든 sem_undo구조체를 추적하여 프로쎄스가 기본 신호기의 계수기안에 명시적인 값을 넣어서 semctl()을 사용할 때 혹은 IPC신호기자원을 제거하려 할 때 구조체에 빠르게 접근할수 있다.

핵심부는 프로쎄스별(per-process), 신호기별(per-semaphore) 목록이 있음으로 하여 이러한 작업을 효률적으로 처리할수 있다. 프로쎄스별목록에서는 취소가능한 연산으로 프로쎄스가 실행한 모든 신호기의 연산을 추적을 유지하고있으며 신호기별목록에서는 취소가능한 연산으로 해당 신호기를 사용한 모든 프로쎄스의 추적을 유지한다. 좀 더 자세히 살펴보면 다음과 같다.

- 프로쎄스별목록에는 프로쎄스가 취소가능한 연산을 수행한 IPC신호기에 대응하는 모든 sem_undo자료구조가 들어있다. 프로쎄스서술자의 semundo마당은 목록의 처음 요소를 가리키며 매 sem_undo자료구조의 proc_next마당은 목록의 다음 요소를 가리킨다.
- 신호기별목록은 신호기에 대해 취소가능한 연산을 수행한 프로쎄스에 대응하는 모든 sem_undo자료구조를 포함한다. sem_array자료구조의 undo마당은 목록의 처음 요소를 가리키고 sem_undo자료구조의 id_next마당은 목록의 다음 요소를 가리킨다.

프로쎄스별목록은 프로쎄스가 완료할 때 사용한다. do_exit()함수가 호출하는 sem_exit()함수는 목록을 확인하면서 프로쎄스가 건드린 모든 IPC신호기의 적절하지 않은 연산을 취소한다. 반면에 신호기별목록은 주로 프로쎄스가 기본신호기에 특정한 값을 대입하기 위해 semctI()함수를 호출할 때 리용한다. 또한 신호기별목록은 IPC신호기를 제거할 때도 리용할수 있다. 관련된 모든 sem_undo자료구조는 semid마당을 -1로 설정하여 무효화한다.

5) 기다리는 요청의 대기렬

핵심부는 배렬에 있는 하나이상의 신호기를 기다리는 프로쎄스를 식별하기 위해 기다리는 요청(pending requests)의 대기렬을 매 IPC신호기에 대응시킨다. 이 대기렬은 sem_queue자료구조의 2중련결목록이며 대기렬의 각 마당은 표 3-21과 같다.

sem_array구조체의 sem_pending과 sem_pending_last마당은 각각 대기렬의 처음과 마지막을 기다리는 요청을 가리킨다. 마지막 마당은 목록을 FIFO처럼 쉽게 처리할수 있게 해준다. 즉 새로운 기다리는 요청은 후에 처리하도록 목록의 마지막에 추가한다. 기다리는 요청의 가장 중요한 마당은 nsops(기다리는 연산에 관련된 기본신호기의 수가 저장되여있다)와 sops(각 신호기연산을 나타내는 정수값배렬을 가리킨다.)이다. sleeper마당에는 연산을 요청한 잠든 프로쎄스의 서술자주소가 저장되여있다.

표 3-21. sem_queue구조체의 마당

형	마 당	설 명
struct sem_queue *	next	다음대기렬요소를 가리키는 지적자
struct sem_queue **	prev	이전대기렬요소를 가리키는 지적자
struct task_struct *	sleeper	신호기연산을 요청한 잠든 프로쎄스지적자
struct sem_undo *	undo	sem_undo구조체의 지적자
int	pid	프로쎄스식별자
int	status	연산완료상태
struct sem_array *	sma	IPC신호기서술자를 가리키는 지적자
int	id	IPC신호기자원의 슬로트색인
struct sembuf *	sops	기다리는 연산의 배렬을 가리키는 지적자
int	nsops	기다리는 연산의 수
int	alter	연산이 신호기값을 변경한다는것을 나타내는 기발

그림 3-15는 기다리는 요청 세개를 가진 IPC신호기를 나타낸다. 두 요청은 취소가능한 연산을 참조하며 semqueue자료구조의 undo마당은 대응하는 sem_undo구조체를 가리킨다.

세번째 기다리는 요청은 대응하는 연산을 취소할수 없으므로 undo마당이 NULL이다.

6) IPC통보문

프로쎄스는 IPC통보문을 리용하여 다른 프로쎄스와 통신을 할수 있다. 프로쎄스가 생성한 각 통보문은 IPC통보문대기렬로 전송되며 다른 프로쎄스가 읽을 때까지 통보문대기렬에 머문다.

통보문은 고정된 크기의 머리부와 가변크기의 본문으로 이루어진다. 통보문에는 통보문류형을 나타내는 정수값이 붙을수 있으며 이러한 정수값을 리용하여 프로쎄스는 통보문대기렬에서 선택적으로 통보문을 읽을수 있다. IPC통보문대기렬에서 프로쎄스가 통보문을 읽으면 핵심부는 통보문을 제거한다. 그러므로 단 한 프로쎄스만이 주어진 통보문을 받게 된다.

프로쎄스는 통보문을 보내기 위해서 msgsnd()함수를 호출하며 다음과 같은것을 변수로 전달한다.

- 목적지 통보문대기렬의 IPC식별자
- 통보문본문의 크기
- 통보문류형과 뒤에 따라오는 통보문본문을 포함하는 사용자방식완충기의 주소

프로쎄스는 통보문을 받기 위해서 msgrcv()함수를 호출하며 다음과 같은것을 변수로 전달한다.

- IPC통보문대기렬자원의 IPC식별자
- 통보문류형과 통보문본문을 복사할 사용자방식완충기의 지적자
- 완충기크기
- 어떤 통보문을 받을것인가를 나타내는 값

t값이 0이면 대기렬에 있는 첫번째 통보문을 반환한다. t가 정수이면 통보문대기렬에 있는 t와 같은 류형의 처음 통보문을 반환한다. 마지막으로 t가 부수이면 함수는 값이 t의 절대값보다 작거나 같은 가장 작은 통보문류형의 처음 통보문을 반환한다.

자원의 소비를 피하기 위해 리용가능한 IPC통보문대기렬자원의 수(기본 16)와 각 통보문크기(기본 8192B), 그리고 대기렬에 있는 통보문의 총 크기(기본 16384B)에 제한이 있다. 그러나 일반적으로 체계관리자는 /proc/sys/kernel/msgmni, /proc/sys/kernel/msgmnb, /proc/sys/kernel/msgmax파일을 수정하여 이러한 값들을 변경할수 있다.

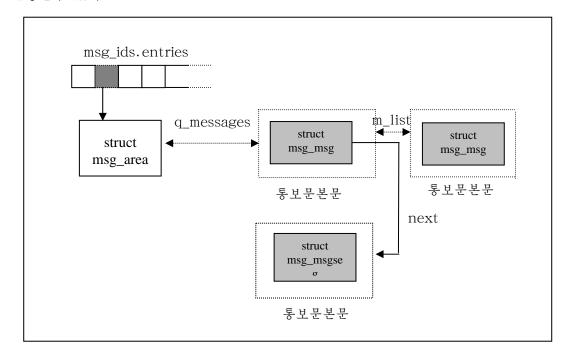



그림 3-16. IPC 통보문대기렬

그림 3-16은 IPC통보문대기렬과 관련있는 자료구조를 보여준다. msg_ids변수에는 IPC통보문대기렬자원류형의 ipc_ids자료구조를 저장한다. entries마당은 mosg_queue 자료구조에 대한 지적자의 배렬이며 각 항목은 IPC통보문대기렬자원 하나를 나타낸다. 형식적으로 배렬은 kern_ipc_perm자료구조의 지적자를 저장하지만 사실 각 구조체는 msg_queue자료구조의 첫번째 마당이다. msg_queue자료구조의 모든 마당은 표 3-22과 같다.

표 3-22. rnsg_queue자료구조

형	마 당	설 명
struct ipc_perm	q_perm	kern_ipc_perm자료구조
long	q_stime	마지막msgsnd()의 시간
long	q_rtime	마지막msgrcv()의 시간
long	q_ctime	마지막변경시간
unsigned long	q_qcbytes	대기렬에 있는 바이트수
unsigned long	q_qnum	대기렬에 있는 통보수
unsigned long	q_qbytes	대기렬의 최대크기(B단위)
int	q_lspid	마지막msgsnd()의 PID
int	q_lrpid	마지막msgrcv()의 PID
struct list_head	q_messages	대기렬에 들어있는 통보문목록
struct list_head	q_receivers	통보문을 받을 프로세스목록
struct list_head	q_senders	통보문을 보낼 프로세스목록

이중에서 가장 중요한 마당은 q_messages이다. 이 마당은 현재대기렬에 있는 모든 통보문을 포함하고있는 2중련결원형목록의 머리부(즉 처음 허수아비요소)를 나타낸다.

각 통보문은 동적으로 할당된 하나이상의 페지에 분할되여 들어간다. 첫번째 페지의 시작부분에 msg_msg형인 자료구조로된 통보문머리부가 저장된다. 이 자료구조의 마당은 표 3-23과 같다. m_list마당에는 대기렬에서의 이전과 다음통보문을 가리키는 지적자를 저장한다. 통보문본문은 msg_msg서술자 바로 다음에서 시작한다. 통보문이 4072B(페지크기-msg_msg 서술자크기)보다 길다면 다음 페지에서 계속되며 이 페지의 주소는 msg_msg서술자의 next마당에 저장된다. 두번째 페지들은 msg_msgseg형서술자로 시작한다. msg_msgseg는 세번째 페지의 주소를 저장하고있는 next지적자만을 담고있다.

표 3-23. msg_msg자료구조

형	마 당	설 명
struct list_head	m_list	통보문목록의 지적자
long	m_type	통보문류형
int	m_ts	통보문본문의 크기
struct msg_msgset *	next	다음 통보문부분의 위치

통보문대기렬이 모두 차게 되면(리용가능한 통보문의 최대수에 도달하거나 전체 크기의 최대값에 도달한 경우) 새로운 통보문을 대기렬에 넣으려는 프로쎄스는 차단된다. msg_queue자료구조의 q_senders마당은 모든 차단된 전송 측 프로쎄스의 서술자를 가리키는 지적자를 담고있는 목록의 머리부이다.

수신측프로쎄스도 통보문대기렬이 비여있으면(또는 프로쎄스가 지정한 류형의 통보문이 대기렬에 없을 때) 차단될수 있다. msg_queue자료구조의 q_receivers마당은 차단된 수신측프로쎄스를 나타내는 msg_receiver자료구조목록의 머리부이다. 매 자료구조는 프로쎄스의 서술자를 가리키는 지적자, 통보문의 msg_msg구조체를 가리키는 지적자, 요청한 통보문 류형을 담고있다.

7) IPC공유기억기

가장 유용한 IPC기구는 공유기억기이다. 공유기억기는 공통자료구조를 IPC 공유기억기령역에 넣으면 둘이상의 프로쎄스가 해당 공통자료구조에 접근할수 있게 해준다. IPC공유기억기령역에 있는 자료구조에 접근하려는 프로쎄스는 IPC공유기억기령역의 페지틀을 사영하는 새로운 기억기령역을 프로쎄스자신의 주소공간에 추가해야 한다. 핵심부는 요구페지화를 통해 이런 페지틀을 쉽게 처리할수 있다. 신호기와 통보문대기렬과 마찬가지로 shmget()함수를 호출하여 공유기억기령역의 IPC식별자를 얻을수 있다. 이미 존재하는 공유기억기령역이 아니라면 새로 생성한다.

shmat()함수를 호출하여 IPC공유기억기령역을 프로쎄스에 붙일수 있다. 이 함수는 IPC공유기억기자원의 식별자를 변수로 받으며 호출한 프로쎄스의 주소공간에 공유기억기령역을 추가한다. 호출한 프로쎄스는 기억기령역에 대한 특정 시작선형주소를 요청할 수 있지만 이 주소는 그다지 중요하지 않으며 공유기억기령역에 접근하려는 각 프로쎄스는 자신만의 주소공간에서 서로 다른 주소를 리용할수 있다. shmat()함수는 프로쎄스의 페지표를 변경하지 않는다. 새로운 기억기령역을 포함하는 페지에 프로쎄스가 접근하려고 할 때 핵심부가 수행하는 일은 뒤에서 설명한다.

shmdt()함수는 IPC식별자가 명시한 IPC기억기령역을 떼여내기 위해 다시 말하면 프로쎄스 주소공간에서 대응하는 기억기령역을 제거하기 위해 호출한다. IPC공유기억기자원은 지속적이라는 점을 다시 상기해보자. 해당 자원을 리용하는 프로쎄스가 없어지고

대응하는 페지가 교체되여 나가더라도 공유기억기령역은 제거할수 없다.

IPC자원의 다른 류형들처럼 사용자방식프로세스가 공유기억기를 너무 많이 사용하는것을 방지하기 위해 IPC공유령역의 수(기본 4096)와 각 토막의 크기(기본 32MB) 그리고 모든 토막의 최대 총 사용량(8GB)제한이 있다.

체계관리자는 /proc/sys/kernel/shmmin과 /proc/sys/kernel/shmmax 그리고 /proc/sys/kernel/shmall파일에 적절한 값을 기록하여 이러한 제한을 조정할수 있다.

IPC공유기억기령역에 관련한 자료구조는 그림 3-17에서 보는바와 같다.

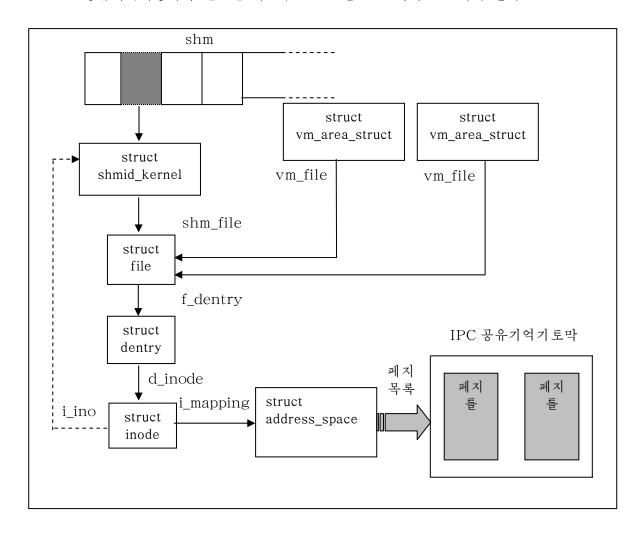



그림 3-17. IPC 공유기억기 자료구조

shm_ids변수는 IPC공유기억기자원류형의 ipc_ids자료구조를 저장한다. entries마당은 IPC공유기억기자원을 나타내는 shmid_kernel자료구조를 가리키는 지적자의 배렬이다. 형식적으로 배렬은 kern_ipc_perm자료구조를 가리키는 지적자를 저장하지만 매

구조체는 단순히 shmid_kernel자료구조의 첫번째 마당이다.

표 3-24는 shmid_kernel자료구조의 모든 마당을 보여준다.

가장 중요한 마당은 shm_file로서 파일객체의 주소를 저장하고있다. 이것은 Linux 2.6의 VFS계층과 IPC공유기억기의 밀접한 통합을 반영한다. 또한 매 IPC공유기억기령역은 shm특수파일체계에 속한 정규파일과 대응한다.

shm파일체계는 체계등록부에 탑재지점이 없기때문에 사용자는 일반적인 VFS체계호출을 리용하여 열거나 접근할수 없다. 그러나 프로쎄스가 토막 하나를 붙일 때마다 핵심부는 do_mmap()을 호출하고 파일의 새로운 공유기억기배치를 프로쎄스의 주소공간에 생성한다. 그러므로 shm특수파일체계에 속한 파일은 하나뿐인 파일객체메쏘드(mmap)를 가지며 이 메쏘드는 shm_mmap()함수로 실현한다.

표 3-24. shmid_kernel구조체의 마당

형	마 당	설 명
struct kern_ipc_perl	shm_perm	kern_ipc_perm자료구조
struct file *	shm_file	토막에 대한 특수파일
int	id	토막에 대한 슬로트색인
unsigned long	shm_nattch	현재 붙어있는 프로쎄스수
unsigned long	shm_segsz	토막크기(B단위)
long	shm_atime	마지막접근시간
long	shm_dtime	마지막 떼여진 시간
long	shm_ctime	마지막변경시간
int	shm_cprid	생성자의 PID
int	shm_lprid	마지막으로 접근한 프로쎄스의 PID

그림 3-17에서 보여주는것처럼 IPC공유기억기령역과 대응하는 기억기령역은 vm_area_struct객체로 표현된다. 이 객체의 vm_file마당은 다시 특수파일의 파일객체를 가리키고 파일객체는 등록부입구점객체와 i마디객체를 가리킨다. i마디의 i_ino마당에 저장된 i마디번호는 실제로는 IPC공유기억기령역의 슬로트색인값이기때문에 i마디객체는 shmid_kernel서술자를 간접적으로 참조하게 된다.

모든 공유기억기령역에 대해서 IPC공유기억기에 속한 페지들은 i마디의 i_mapping마당을 통해 참조하는 address_space객체형태로 페지캐쉬에 포함된다.

8) IPC공유기억기령역의 페지를 교환하여 내보내기

핵심부는 공유기억기령역에 포함된 페지를 교환하여 내보낼 때 매우 주의하여야 하며 교환기캐쉬의 역할이 매우 중요하다.

《try_to_swap_out()함수》에서 설명하겠지만 핵심부는 address_space 객체가 소

유한 페지를 교환하여 내보내기 위해서 페지에 불결한(dirty)것이라고 표시하여 디스크로 자료를 전송하도록 한 다음 프로쎄스의 페지표에서 페지를 제거한다. 페지가 공유파일기억기배치에 포함되여있다면 결국 페지를 더는 어떤 프로쎄스도 참조하지 않게 되고 shrink_cache()함수는 이 페지를 해제하여 형제체계으로 돌려보낸다. 이것은 페지에 있는 자료가 디스크에 있는 자료의 복사본이므로 정확히 동작한것이라고 볼수 있다.

그러나 IPC공유기억기령역의 페지는 디스크에 영상이 없는 특수한 i마디로 배치한다. 그리고 IPC 공유기억기는 지속적이며 따라서 공유기억기의 페지는 토막이 아무 프로쎄스에도 붙어있지 않더라도 계속 보호해야 한다. 그러므로 핵심부는 대응하는 페지틀을 회수할 때 페지를 간단히 제거할수 없다. 대신 페지를 교환하여 내보내야 한다.

try_to_swap_out()함수는 이러한 특별한 경우를 검사하지 않기때문에 령역에 속한 페지는 불결한것으로 표시되며 프로쎄스주소공간에서 제거된다. 페지캐쉬에서 최근에 사용하지 않은 페지를 주기적으로 제거하는 shrink_cache()함수도 이러한 특별한 경우에 대한 검사기능이 없으므로 결국 소유자인 address_space 객체의 writepage 메쏘드를 호출한다.

이제 공유기억기페지가 교체되여 나갈 때 IPC공유기억기를 보호하는 방법을 보자.

IPC 공유기억기령역에 속한 페지는 변형된 shmem_writepage()함수를 리용하여 writepage메쏘드를 실현한다. shmem_writepage()함수는 교환령역에 새로운 페지슬로트를 생성하고 페지캐쉬에서 교환캐쉬로 페지를 옮긴다.(페지의 소유자인 address_space 객체의 변경일뿐이다.)

함수는 또한 교체되여나간 페지의 식별자를 i마디객체의 파일체계별령역의 shmem_inode_info 구조체에 저장한다. 페지는 교환공간에 즉시 기록되지 않으며 shink_cache()함수가 다시 호출할 때 기록된다.

9) IPC공유기억기령역에 대한 요구페지화

shmat()함수를 통해 프로쎄스에 추가된 페지는 허수아비페지이다. 함수는 새로운 기억기령역을 프로쎄스주소공간에 추가하지만 프로쎄스페지표는 갱신하지 않는다. IPC 기억기령역의 페지는 교체되여 나갈수도 있다. 그러므로 이러한 페지는 요구페지화기구로 처리된다.

페지오유(page fault)는 프로쎄스가 아직 페지틀을 할당하지 않은 IPC공유기억기령역의 위치에 접근하려고 시도할 때 발생한다. 이와 대응하는 례외조종기는 잘못된 주소가 프로쎄스주소공간내부에 있는지 그리고 대응하는 페지표입구점이 NULL인가를 결정한 후 do_no_page()함수를 호출한다. 이 함수는 기억기령역에 nopage메쏘드가 정의되여있는가를 검사한 다음 이 메쏘드를 호출하고 페지표입구점을 메쏘드가 반환한 값으로 설정한다.

IPC공유기억기에 사용하는 기억기령역은 언제나 nopage메쏘드를 정의한다. 이 메쏘드는 shmem_nopage()함수로 실현되여있으며 다음연산을 수행한다.

1. VFS객체들의 내부지적자사슬을 훑으면서 IPC기억기자원의 i마디객체의 주소를

가져온다.(그림 3-17 참고)

2. 기령역서술자의 vm_start마당과 요청한 주소로부터 토막안에서의 론리적페지번호를 계산한다.

3. 페지가 이미 교환캐쉬에 포함되여있는지 확인한다. 만약 포함되여있다면 주소를 반환한 후 완료한다.

4. i마디객체내부의 shmem_inode_info가 론리적페지번호수자에 대한 교체되여 나간 페지식별자를 저장하고있는지 확인한다. 만약 저장되여있으면 swapin_readahead()를 호출하여 교체들여오기(Swap-In)연산을 진행한다. 자료전송이 완료될 때까지 대기한 후 페지주소를 반환함으로써 완료한다.

5. 그외 경우에는 교환령역에 페지가 저장되여있지 않다. 함수는 형제체계에서 새로운 페지를 할당하고 이 페지를 페지캐쉬에 추가한 다음 그 주소를 반환한다.

do_no_page()함수는 프로쎄스의 페지표에 있는 잘못된 주소에 대응하는 항목이 메쏘드가 반환한 페지틀을 가리키도록 설정한다.

제 4 장. 기억기

제 1 절. 기억기주소지정

이 절에서는 주소를 지정하는 방법에 대하여 서술한다. 일반적으로 조작체계가 물리적인 기억기에 대한 모든 사항을 알고있어야 할 필요는 없다. 최근의 극소형처리기에는 기억기관리를 더 효률적으로 할수 있도록 프로그람작성오유에 적극적으로 대응하는 하드웨어회로가 여러개 들어있다.

이 절에서는 80x86극소형처리기에서 기억기소편의 주소를 지정하는 방법과 Linux에서 하드웨어에서 제공하는 주소지정회로를 활용하는 방법에 대하여 자세히 서술한다. 가장 보편적인 Linux기반에서 이것을 구현하는 방법을 자세히 알게 되면 일반적인 페지화에 대한 리론과 다른 기반에서 구현하는 방법을 더 쉽게 리해할수 있다.

1. 기억기주소

프로그람작성자들은 기억기주소(memory address)를 단순히 기억기요소(cell) 내용에 접근하는 수단으로 생각한다. 그러나 80x86극소형처리소자를 다루려면 다음과 같은 세 종류의 주소를 서로 구별할수 있어야 한다.

1) 론리주소

기계어명령에서 피연산자나 명령의 주소를 지정할 때 사용한다. 이 류형의 주소는 MS-DOS와 Windows프로그람작성자들이 프로그람을 작성할 때 프로그람을 여러 토막으로 쪼개도록 하는 유명한 Intel의 토막구조를 구체화한것이다. 론리주소(logical address)는 토막과 토막의 시작부터 실제주소까지의 거리를 나타내는 편위(offset, displacement라고도 한다.)로 이루어진다.

2) 선형주소

부호가 없는 32bit 크기의 정수값으로 4GB 즉 기억기요소 4,294,967,296개까지 주소를 지정할수 있다. 선형주소(가상주소라고도 한다.)는 보통 16진수로 표시하며 값의 범위는 0x00000000에서 0xffffffff까지이다.

3) 물리주소

기억기소편에 들어있는 기억기요소를 지정하는데 사용하는 주소로 극소형처리기에서 주소단자(address pin)를 통해 기억기모선(memory bus)에 보내는 전기신호에 해당한다. 물리주소(physical address)는 부호가 없는 32bit크기의 옹근수로 나타낸다.

CPU의 조종장치는 토막유니트(segmentation unit)라는 하드웨어회로를 리용하여 론리주소를 선형주소로 변환하고 계속해서 페지화유니트(paging unit)라는 또 다른 하드웨어회로를 리용하여 선형주소를 물리주소로 변환한다.(그림 4-1 참고)

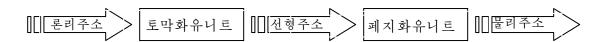



그림 4-1. 론리주소의 변환과정

다중과제처리체계에서는 모든 CPU가 같은 기억기를 공유한다. 즉 서로 독립적인 CPU들이 동시에 기억기소편에 접근할수도 있다. 기억기소편에 읽고쓰는 동작은 차례대로 이루어져야 하므로 모선과 모든 기억기소편사이에는 기억기중재자(memory arbiter)라는 하드웨어회로가 들어간다. 이 회로의 역할은 소편을 사용하지 않고있으면 CPU가 소편에 접근할수 있도록 하고 다른 처리기의 요청을 처리하는중이면 소편에 대한 접근을 지연하는것이다. 단일처리기체계에서도 기억기중재자를 사용하는데 CPU와 별개로 동작하는 DMA라는 전용소자가 있기때문이다.

다중처리기체계에서는 입력포구가 더 많기때문에 기억기중재자의 구조가 훨씬 복잡하다. 례를 들어 펜티움 CPU 두개를 사용하는 경우 각 소편의 입구에 포구가 두개인 중재자를 둔 후 두 CPU가 공통모선을 사용하려 할 때 먼저 동기화통보문을 교환하도록 한다. 중재자는 하드웨어회로로 이루어지기때문에 프로그람작성견지에서는 보이지 않는다.

2. 하드웨어토막화

Intel극소형처리기는 80386모형부터 실방식(real mode)과 보호방식(protected mode)이라는 서로 다른 두가지 방법으로 주소변환을 하기 시작하였다. 이 두가지 방식은 아래에서 설명한다. 실방식은 주로 이전모형과 처리기호환성을 유지하고 조작체계가 기동할수 있도록 하기 위한것이다.

1) 토막등록기

론리주소는 두부분 즉 토막식별자(segment identifier)와 토막내에서 상대적인 주소를 나타내는 편위(offset)로 구성된다. 토막식별자는 토막선택기(segment selector)라는 16bit마당이며 편위는 32bit마당이다.

처리기는 토막선택기를 빨리 얻어오려고 토막선택기를 보관할 목적으로만 사용하는 토막등록기(segmentation)를 제공한다. 이 등록기로는 cs, ss, ds, es, fs, gs가 있다.

비록 등록기가 여섯개밖에 없지만 프로그람은 등록기의 내용을 저장하고 후에 다시 복구하는 방법을 통해 똑같은 토막등록기를 다른 용도로 재리용할수 있다.

토막등록기 6개중 3개는 특별한 용도로 사용한다.

- cs 코드토막등록기(code segment register)로서 프로그람명령을 담고있는 토막을 가리킨다.
- ss 탄창토막등록기(stack segment register)로서 현재프로그람의 탄창을 담고있는 토막을 가리킨다.

- ds 자료토막등록기(data segment register)로서 정적인 자료와 대역자료를 담고있는 토막을 가리킨다.

나머지 세 토막등록기는 범용으로 임의의 토막을 가리킬수 있다.

cs등록기에는 중요한 기능이 하나 더 있다. 여기에는 CPU의 현재특권준위(CPL: Current Privilege Level)를 나타내는 2bit마당이 있다. 0은 가장 높은 특권준위를, 3은 가장 낮은 특권준위을 나타낸다. Linux는 0과 3준위만을 사용하는데 각각 핵심부방식(kenel mode)과 사용자방식(user mode)이라고 한다.

2) 토막서술자

토막의 특징을 서술하는 8B 크기의 토막서술자(segment descriptor)로서 각 토막를 표현한다.(그림 4-2 참고)

토막서술자는 대역서술자표 (GDT:Global Descriptor Table)나 국부서술자표(LDT:Local Descriptor Table)에 저장된다.

자료토막서술자

63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35

BASE(24-31)	G	B	O	A V L	LIMIT (16-19)	1	D P L	S = 1	TYPE	BASE(16-23)
BASE(0-15)						LIMIT(0-15)				

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

코드토막서술자

63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32

BASE(24-31)	G	B	O	A V L	LIMIT (16-19)	1	D P L	S = 1	TYPE	BASE(16-23)
BASE(0-15)						LIMIT(0-15)				

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

체계토막서술자

63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32

<table>
<tr><td>BASE(24-31)</td><td>G</td><td>B</td><td>O</td><td>A
V
L</td><td>LIMIT
(16-19)</td><td>1</td><td>D
P
L</td><td>S
=
0</td><td>TYPE</td><td>BASE(16-23)</td></tr>
<tr><td colspan="6">BASE(0-15)</td><td colspan="5">LIMIT(0-15)</td></tr>
</table>

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

그림 4-2. 토막서술자형식

보통 GDT는 하나만 정의한다. 반면에 각 프로쎄스는 GDT에 들어있는 토막외에 토막이 더 필요한 경우 자신만의 LDT를 가질수 있다. 주기억기에서 GDT가 위치한 주소는 gdtr처리기 등록기에 현재 사용하는 LDT의 주소는 ldtr처리기등록기에 들어있다.

각 토막서술자는 다음 마당으로 구성된다.

- 토막이 시작하는 선형주소를 담는 32bit Base마당
- 파립도(granularity)를 나타내는 G기발, 이 기발이 0이면 토막크기는 B단위이고 1이면 4096B를 곱한 크기가 된다.
- 토막길이를 지정하는 20bit Limit마당 G가 0이면 토막크기는 1B에서 1MB까지 될수 있고 G가 1이면 4kB에서 4GB까지 될수 있다.
- 체계기발 S가 0이면 토막은 핵심부자료구조체를 포함하는 체계토막이고 1이면 일반적인 코드나 자료를 포함하는 토막이다.
- 토막의 종류와 접근권한을 나타내는 4bit크기의 Type마당

많이 사용하는 토막서술자종류는 다음과 같다.

○ **코드토막서술자**

토막서술자는 코드토막을 나타낸다. GDT나 LDT 어디에나 있을수 있다.

이 서술자는 S기발은 1로 설정된다.

○ **자료토막서술자**

토막서술자는 자료토막을 나타낸다. GDT나 LDT 어디에나 있을수 있다. 이 서술자의 S기발은 1로 설정된다. 탄창토막은 일반 자료토막으로 실현한다.

○ **작업상태토막서술자**

토막서술자는 처리기등록기의 내용을 저장하기 위해 사용하는 작업상태토막(TSS : Task State Segment)을 나타낸다.

GDT에만 있을수 있다. 해당 프로쎄스가 현재CPU에서 실행중인가 하는 여부에 따라 Type마당의 값이 11이나 9가 된다. 이 서술자의 S기발은 0으로 설정한다.

○ **국부서술자표서술자**

토막서술자는 LDT를 포함하는 토막을 나타낸다. GDT에만 있을수 있으며 Type마

당의 값은 2이다. 이 서술자의 S기발은 0으로 설정한다. 다음 80x86처리기에서 토막서술자가 GDT에 있는지 아니면 프로쎄스의 LDT에 있는지 알아내는 방법을 설명한다.

- 토막에 대한 접근을 제한하는데 사용하는 2bit크기의 서술자특권준위(DPL:descriptor Pivilege Level)마당

이 마당은 해당 토막에 접근하는데 필요한 최소 CPU특권준위를 나타낸다. 따라서 DPL이 0인 토막은 현재특권준위(CPL)가 0일 때 즉 핵심부방식에서만 접근할수 있다. 한편 DPL이 3인 토막은 모든 CPL에서 접근할수 있다.

- Segment-Present기발로서 토막이 현재 주기억기에 없으면 0으로 설정한다. Linux에는 토막을 통채로 디스크에 교체하여 넣지 않기때문에 이 마당을 항상 1로 설정한다.
- 토막이 코드를 포함하는지 자료를 포함하는가에 따라 D와 B기발이 추가된다.

이 기발의 의미는 두 경우에 따라 조금 다른데 기본적으로는 토막편위를 나타내는데 사용하는 주소가 32bit면 1로, 16bit면 0으로 설정한다.

- 예약된 비트(53)는 항상 0으로 설정한다.
- 조작체계가 사용할수 있는 AVL기발. Linux에서는 이 기발을 무시한다.

3) 토막서술자에로의 빠른 접근

앞에서 론리주소는 16bit크기의 토막선택기와 32bit크기의 편위로 구별되며 토막등록기는 토막선택기만 저장한다고 언급하였다.

80x86처리기는 론리주소를 선형주소로 빠르게 변환하려고 프로그람화가능한 토막등록기 여섯개 각각에 대해 프로그람불가능한 등록기(nonprogrammable register) 즉 프로그람작성자가 설정할수 없는 등록기를 추가로 제공한다.

이 프로그람화가 불가능한 등록기는 각각 자신과 관련된 토막등록기에 들어있는 토막선택기가 지정한 8B 크기의 토막서술자를 포함한다. 토막선택기를 토막등록기에 적재할 때마다 해당 토막서술자를 기억기에서 꺼내여 대응하는 프로그람불가능한 등록기에 적재한다. 이때부터 이 토막을 참조하는 론리주소를 주기억기에 저장된 GDT나 LDT에 접근하지 않고도 변환할수 있다. 처리기는 토막서술자를 포함한 CPU등록기를 직접 참조할수 있기때문이다. 토막등록기의 내용이 바뀔 때에만 GDT나 LDT에 접근할 필요가 있다.(그림 4-3 참고) 각 토막선택기는 다음과 같은 마당을 포함한다.

- GDT나 LDT에 들어있는 토막서술자입구를 가리키는 13bit크기의 색인
- 표지시자(TI:Table Indicator)기발

토막서술자가 GDT에 있는지(TI=0), LDT에 있는지 (TI=1)를 나타낸다.

- 2bit크기의 요청특권준위(RPL:Requested Privllege Level)마당

해당 토막선택기를 cs등록기에 적재하면 이 값이 현재특권준위(Current Privilege Level)가 된다.

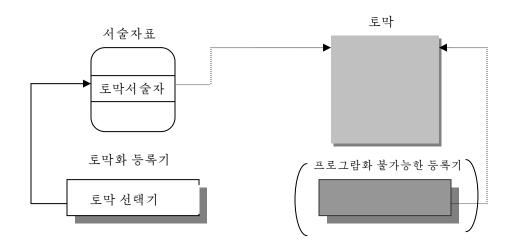



그림 4-3. 토막선택기와 토막서술자

토막서술자는 8B크기이므로 토막선택기의 웃자리 13bit색인값에 8을 곱해서 GDT나 LDT에 있는 상대주소를 계산할수 있다. 례를 들어 GDT가 0x00020000에 있고 토막선택기가 가리키는 색인이 2이면 해당하는 토막서술자의 주소는 0x00020000+(2x8), 즉 0x00020010이 된다.

GDT의 첫번째 입구는 항상 0으로 설정한다. 이것은 토막선택기로 0을 지정하는 론리주소를 잘못된 주소로 만들어 처리기례외를 발생시키도록 한다. GDT에 저장할수 있는 최대토막서술자의 개수는 8191, 즉 $2^{13}-1$개이다.

RPL마당은 자료토막에 접근할 때 처리기 특권준위을 선택적으로 약화시키는데 사용할수도 있다.

빈 지적자는 프로그람작성에서 흔히 하는 실수이다. 특히 지적자와 기억기할당에 관련해서 많이 일어난다. 따라서 핵심부코드에 있을수 있는 이런 실수를 감축할수 있도록 주소공간의 시작부분을 사용하지 못하게 만들어서 이 령역에 접근하는 경우 례외를 발생하게 만드는것이다.

4) 토막화유니트

그림 4-4는 론리주소를 선형주소로 변환하는 과정을 자세히 보여준다. 토막화유니트는 다음과 같은 일을 한다.

- 토막선택기의 TI마당을 검사하여 토막서술자가 어떤 서술자표에 들어있는가를 확인한다. TI마당은 서술자가 GDT에 있는지(이 경우 토막화유니트는 gdtr등록기에서 GDT의 시작선형주소를 가져온다.) 아니면 현재 활성화된 LDT에 있는지(이 경우 토막화유니트는 ldtr등록기에서 LDT의 시작선형주소를 가져온다.)를 나타낸다.
- 토막선택기의 Index마당으로 토막서술자의 주소를 계산한다. Index마당에 8(토막서술자의 크기)을 곱한 값을 gdtr이나 ldtr등록기의 내용에 더한다.

• 토막서술자의 Base마당에 론리주소의 편위를 더해서 선형주소를 얻는다.

토막등록기와 관련된 프로그람불가능한 등록기로 하여 앞의 두 작업은 토막등록기가 바뀐 경우에만 수행하면 된다.

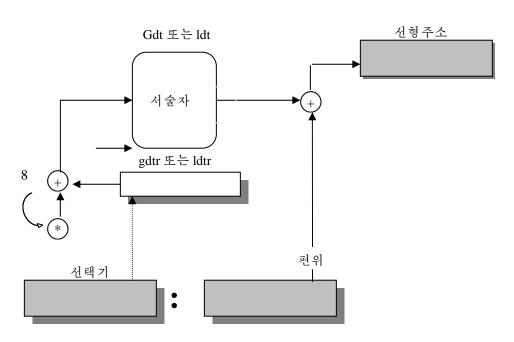



그림 4-4. 론리주소의 변환과정

3. Linux에서의 토막화

80x86극소형처리기는 프로그람작성자가 응용프로그람을 보조루틴이나 대역자료령역, 국부자료령역 같은 론리적인 부분으로 쪼갤수 있도록 토막화라는 기법을 지원한다.

그러나 Linux는 토막화를 매우 제한적으로 사용한다. 사실 토막화와 페지화는 둘다 프로쎄스의 물리주소공간을 쪼개는데 사용하기때문에 어느 정도 중복되는 측면도 있다. 토막화가 각 프로쎄스에 다른 선형주소공간을 할당하는데 대해 페지화는 똑같은 선형주소공간을 다른 물리주소공간과 배치해준다. Linux는 다음과 같은 리유때문에 토막화보다는 페지화를 먼저 호출한다.

• 모든 프로쎄스가 똑같은 토막등록기값을 가지면 즉 모든 프로쎄스가 똑같은 선형주소공간을 공유하면 기억기관리가 더 간단해진다.

• Linux설계의 목적중 하나는 일반적인 다른 구조(architecture)와 호환하게 하는것인데 RISC구조에서는 토막화를 매우 제한적으로 지원한다.

Linux에서는 80x86구조에서 필요로 하는 경우에만 토막화를 사용한다. 모든 프로쎄스가 똑같은 론리주소를 사용하기때문에 정의해야 하는 토막의 개수는 매우 적으며 모

든 토막서술자를 대역서술자표(GDT)에 저장할수 있다. GDT는 gdt_table이라는 배렬로 구성되여있고 gdt라는 변수로서 참조한다.

이러한 기호는 《arch/1386/kernel/heads》파일에서 정의하고있다.

핵심부는 국부서술자표(LDT)를 사용하지 않지만 프로쎄스가 자기만의 LDT를 만들수 있는 modify_ldt()체계호출을 제공한다. 이것은 토막기반의 MS Windows용 응용프로그람을 실행하는 Wine과 같은 응용프로그람에 유용하다.

다음은 Linux에서 사용하는 토막의 목록이다.

- 핵심부코드토막 GDT에 있는 이 토막서술자의 매 마당값은 다음과 같다.
    - Base = 0x00000000
    - Limit = 0xfffff
    - G(파립도기발) = 1, 토막크기는 페지단위이다.
    - S(체계기발) = 1, 일반코드나 자료토막이다.
    - Type = 0xa, 읽기 및 쓰기 가능한 자료토막이다.
    - DPL(서술자특권준위) = 0, 핵심부방식을 나타낸다.
    - D/B(32bit주소기발) = 1, 32bit편위주소를 나타낸다.

따라서 이 토막의 선형주소범위는 0에서 $2^{32}-1$이다. S와 Type마당은 읽고 실행할수 있는 코드토막이라는것을 나타낸다. DPL값이 0이므로 핵심부방식에서만 접근할수 있다.

이에 해당하는 토막선택기는 __KERNEL_CS마크로로 정의되여있다. 핵심부가 이 토막을 가리키고 싶을 때에는 마크로가 만들어내는 값을 cs등록기에 저장만 하면 된다.

- 핵심부자료토막 GDT에 있는 해당 토막서술자의 매 마당값은 다음과 같다.
    - Base=0x00000000
    - Limit=0xfffff
    - G(파립도기발)=1, 토막크기는 페지단위이다.
    - S(체계기발)=1, 일반코드나 자료토막이다.
    - Type=2, 읽기 및 쓰기 가능한 자료토막이다.
    - DPL(서술자특권준위)=0, 핵심부방식을 나타낸다.
    - D/B(32bit 주소기발)=1, 32bit 편위주소를 나타낸다.

이 토막은 읽고 쓸수있는 자료토막임을 나타내는 Type마당을 제외하고 핵심부코드토막과 동일하다.(사실 이것들의 선형주소공간도 서로 겹친다.) 이에 해당하는 토막선택기는 __KERNEL_DS마크로로 정의하고있다.

- 사용자방식에 있는 모든 프로쎄스가 공유하는 사용자코드토막

GDT에 있는 해당 토막서술자의 각 마당값은 다음과 같다.

- Base=0x00000000
- Limit=0xfffff

- G(과립도 기발)=1, 토막크기는 페지단위이다.
- S(체계 기발)=1, 일반코드나 자료토막이다.
- Type=0xa, 읽기 및 실행 가능한 코드토막이다.
- DPL(서술자특권준위)=3, 사용자방식을 나타낸다.
- D/B(32bit주소기발)=1, 32bit편위주소를 나타낸다.

S와 DPL마당은 이 토막이 체계토막이 아니며 특권준위가 3이라는것을 나타낸다. 따라서 핵심부방식과 사용자방식에서 모두 이 토막에 접근할수 있다. 해당 토막선택기는 __USER_CS마크로로 정의하고있다.

○ 사용자방식에 있는 모든 프로쎄스가 공유하는 사용자자료토막

GDT에 있는 해당 토막서술자의 매 마당값은 다음과 같다.

- Base=0x00000000
- Limit = 0xfffff
- G(과립도기발)=1, 토막크기는 페지단위이다.
- S(체계기발)=1, 보통 코드나 자료토막이다.
- Type=2, 읽기 및 쓰기 가능한 자료토막이다.
- DPL(서술자특권준위)=3, 사용자방식을 나타낸다.
- D/B(32bit주소기발)=1, 32bit편위주소를 나타낸다.

이 토막은 앞서 나온 사용자코드토막과 겹친다. 두개는 Type값만 제외하고 동일하다. 해당 토막선택기는 __USER_DS 마크로로 정의하고있다.

○ 각 처리기별 작업상태토막(TSS)

각 TSS에 해당하는 선형주소공간은 핵심부자료토막에 해당하는 선형주소공간의 일부이다. 모든 TSS는 init_tss배렬에 차례로 저장된다.

n번째 CPU용 TSS서술자의 Base마당은 init_tss배렬의 n번째 원소를 가리킨다. TSS토막의 크기가 236B이기때문에 G(과립도)기발은 0으로, Limit기발은 0xeb로 설정된다. Type마당은 9나 11로 설정되고(유효한 32bit TSS) 사용자방식프로쎄스가 TSS토막에 접근하면 안되기때문에 DPL은 0으로 설정된다.

○ 보통 모든 프로쎄스가 공유하는 기정 국부서술자표(default LDT).

이 토막은 default_ldt변수에 들어있다. 기정LDT에는 빈 토막서술자(null segment descriptor)로 이루어진 입구 하나만 들어있다. 각 처리기마다 자기만의 LDT토막서술자가 있는데 이것은 보통 공용으로 사용하는 기본 LDT토막을 가리킨다. 이 토막서술자의 Base마당은 default_ldt의 주소로, Limit마당은 7로 설정한다. 실제로 LDT를 필요로 하는 프로쎄스를 실행할 때에는 GDT에서 이 프로쎄스를 실행하는 CPU에 해당하는 LDT서술자를 프로쎄스가 만든 LDT에 대한 서술자로 대체한다.

○ 고급전원관리(APM:Advanced Power Management)지원과 관련한 토막 4개

APM은 체계의 전원을 관리하는 BIOS함수들로 이루어진다. 핵심부가 APM을 지

원하면 GDT에 있는 4개의 입구는 APM관련핵심부함수를 포함하는 두 자료토막 및 두 코드토막의 서술자를 저장한다.

리눅스의 GDT	토막선택기
빈값(null)	0x00
사용하지 않음	
핵심부 코드	0x10 (__KERNEL_CS)
핵심부 자료	0x18 (__KERNEL_DS)
사용자 코드	0x20 (__USER_CS)
사용자 자료	0x28 (__USER_DS)
사용하지 않음	
사용하지 않음	
오유있는 BIOS의 APM	0x40
APM코드	0x48
APM16bit코드	0x50
APM자료	0x58
CPU-0 TSS	
CPU-0 LDT	
사용하지 않음	
사용하지 않음	
CPU-1 TSS	
CPU-1 LDT	
사용하지 않음	
사용하지 않음	
…	

그림 4-5. 대역서술자표(GDT)

결국 그림 4-5에서 보여주는것처럼 GDT는 몇가지 공용서술자들과 체계에 있는 각 CPU용 토막서술자 한쌍(하나는 TSS토막용, 다른 하나는 LDT토막용)을 포함한다. 효률적으로 동작하기 위해 GDT에 있는 몇개의 입구는 사용하지 않으며 이것은 보통과 같이 접근하는 토막서술자들이 똑같은 32B 크기의 하드웨어캐쉬선로에 있게 한다.

앞에서 설명했지만 cs등록기에 저장된 토막선택기의 RPL마당은 처리기가 사용자방식에 있는지 핵심부방식에 있는지를 나타내는 CPU의 현재 특권준위(CPL:Current

교육성 교육정보쎈터

Privilege level)를 지정한다. CPL이 바뀔 때 일부 토막등록기도 갱신되여야 한다. 례를 들어 CPL이 3(사용자방식)일 때 ds등록기는 사용자자료토막의 토막선택기를 포함해야 하지만 CPL이 0일 때에는 핵심부자료토막의 토막선택기를 포함해야 한다.

ss등록기도 이와 비슷하다. 이 등록기는 CPL이 3일 때 사용자자료토막안에 존재하는 사용자방식탄창을 참조해야 하지만 CPL이 0일 때에는 핵심부자료토막안에 존재하는 핵심부방식탄창을 참조해야 한다. Linux는 사용자방식에서 핵심부방식으로 전환할 때 항상 ss등록기가 핵심부자료토막의 토막선택기를 포함하는가를 확인한다.

4. 하드웨어페지화

페지화유니트(paging unit)는 선형주소를 물리주소로 변환한다. 페지화유니트는 요청한 접근종류가 해당 선형주소의 접근권한에 맞는가를 검사한다. 기억기에 잘못 접근한 경우에는 페지오유새치기(page fault exception)가 발생한다.

효률적인 관리를 위해 선형주소를 페지(page)라는 고정된 크기로 나눈다. 한 페지에 있는 련속된 선형주소는 련속된 물리주소로 배치된다. 이런식으로 핵심부는 페지내의 모든 선형주소마다 물리주소와 접근권한을 지정하지 않고 페지마다 지정한다. 일반적인 관습대로 선형주소집합을 가리키거나 이 주소그룹에 들어있는 자료를 가리키는 경우 모두 《페지》라는 용어를 사용하기로 한다.

페지화유니트는 RAM의 모든 령역이 페지틀(page frame), 물리페지(physical page)라는 고정된 길이로 나뉘어있다고 생각한다. 매 페지틀마다 페지가 하나씩 들어간다. 즉 페지틀의 크기와 페지크기는 일치한다. 페지틀은 주기억기의 구성요소이기때문에 저장령역이라고 할수 있다. 그러나 페지와 페지틀을 구별할수 있어야 한다. 페지는 단순히 자료블로크로서 어느 페지틀이나 디스크에도 저장할수 있다.

선형주소를 물리주소로 배치하는 자료구조를 페지표(page table)라고 한다. 페지표는 주기억기에 있으며 핵심부는 페지화유니트를 사용하기 전에 페지표를 정확히 초기화해야 한다.

80x86처리기에서는 조종등록기 cr0의 PG기발을 설정하여 페지화를 가능하게 한다. PG=0이면 선형주소는 곧바로 물리주소가 된다.

1) 정규페지화

80386부터 Intel처리기의 페지화유니트는 4kB크기의 페지를 사용한다.

32bit크기인 선형주소는 다음 세 마당으로 나누어진다.

등록부

웃자리 10bit

표

중간 10bit

편위

마지막 12bit

선형주소변환은 두단계로 이루어지며 매 단계마다 일종의 변환표를 사용한다. 첫째 단계에서 사용하는 변환표를 페지등록부(Page Directory)라고 하고 둘째단계에서 사용하는 표를 페지표(PageTable)라고 한다. 이렇게 두 단계를 거치는 리유는 프로쎄스마다 필요한 페지표가 RAM에서 차지하는 크기를 줄이기 위해서이다. 간단하게 한단계 페지표를 사용한다고 하면 매 프로쎄스마다 입구 2^{20}개를 포함하는(매 입구는 4B크기이므로 4MB RAM이 필요하다.) 페지표가 필요할것이다.(프로쎄스가 4GB의 선형주소공간을 모두 사용하는 경우) 두 단계 페지표를 사용하면 실제로 프로쎄스가 사용하는 가상기억기령역에 대해서만 페지표가 필요하므로 기억기를 절약할수 있다.

동작중인 모든 프로쎄스는 자기의 페지등록부를 가져야 한다. 그렇지만 한번에 프로쎄스의 모든 페지표용으로 기억기를 할당할 필요는 없다. 실제 프로쎄스가 필요로 할 때에만 페지표용기억기를 할당하는것이 더 효률적이다.

현재 사용중인 페지등록부의 물리주소는 조종등록기 cr3에 들어있다. 선형주소내의 등록부마당은 해당 페지표를 가리키는 페지등록부의 입구를 결정한다. 주소의 표마당은 해당 페지를 포함하는 페지틀의 물리주소를 담은 페지표의 입구를 결정한다. 편위마당은 페지틀에서 상대위치를 나타낸다.(그림 4-6 참고) 편위의 길이는 12bit이기때문에 각 페지는 4096B의 자료를 가진다.

등록부와 표 마당은 모두 10bit이다. 따라서 페지등록부와 페지표는 각각 1024개의 입구를 포함할수 있다. 32bit주소에서 짐작하는것처럼 페지등록부 하나로 기억기요소주소 1024*1024*4096=2^{32}개를 지정할수 있다.

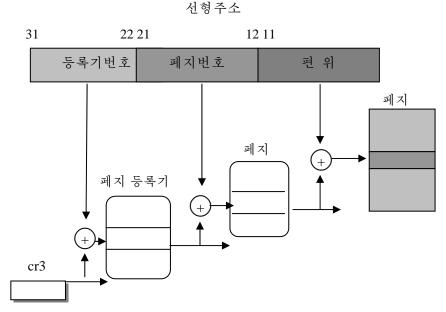



그림 4-6. 80x86 처리기의 페지화

페지등록부와 페지표의 입구구조는 똑같다. 매 입구에는 다음과 같은 마당이 있다.

- **Present기발**

이 기발이 1이면 이 입구가 참조하는 페지(또는 페지표)는 주기억기에 있다. 이 기발이 0이면 페지가 주기억기에 없으며 입구에 있는 다른 비트는 조작체계의 필요에 따라 다른 용도로 사용할수 있다. 주소변환을 하는데 필요한 페지표나 페지등록부입구의 Present기발이 0인 경우 페지화유니트는 선형주소를 조종등록기 cr2에 저장한 후 례외번호 14번 즉 페지오유 례외를 발생시킨다.

- **페지틀물리주소의 웃자리 20bit를 포함한 마당**

각 페지틀의 용량은 4kB이기때문에 페지틀의 물리주소는 4096의 배수여야 한다. 따라서 물리주소의 아래자리 12bit는 항상 0이다. 이 마당이 페지등록부를 참조하는 경우 페지틀에 페지표가 들어가며 페지표를 참조하는 경우에는 자료가 있는 페지가 들어간다.

- **Accessed기발**

페지화유니트는 해당 페지틀에 접근할 때마다 이 기발을 1로 설정한다. 조작체계는 이 기발을 교환하여 내보내기(swap out)할 페지를 선택하는데 사용할수 있다. 페지화유니트는 이 기발을 절대로 지우지 않는데 이것은 조작체계의 몫이다.

- **Dirty기발**

페지표입구에만 적용된다. 페지화유니트는 페지틀에 쓰기작업을 수행할 때마다 이 기발을 1로 설정한다. Accessed기발과 마찬가지로 조작체계는 이 기발을 바꾸어 내보내기할 페지를 결정하는데 사용할수 있다. 페지화유니트는 이 기발을 절대로 지우지 않으며 이것은 조작체계가 할 일이다.

- **Read/Write기발**

이 기발에는 페지나 페지표의 접근권한(읽기/쓰기 또는 읽기)이 있다.

- **Usef/Supervisor기발**

페지나 페지표에 접근하는데 필요한 특권(privilege)준위를 나타낸다.

- **PCD와 PWT기발**

하드웨어캐쉬가 페지나 페지표를 다루는 방법을 조종한다.

- **Page Size기발**

페지등록부입구에만 해당한다. 이 기발이 1이면 입구는 2MB 또는 4MB 크기의 페지틀을 가리킨다.

- **Global기발**

페지등록부입구에만 해당한다. 이 기발은 펜티움프로 CPU에서 등장했으며 자주 사용하는 페지가 TLB캐쉬에서 사라지는것을 막기 위한것이다. 등록기 cr4의 PGE(Page Global Enable)기발이 설정된 경우에만 동작한다.

2)확장페지화

80x86 극소형처리기는 펜티움모형부터 페지틀의 크기가 4kB 대신 4MB가 될수 있

는 《확장페지화(extended paging)》를 도입하였다.(그림 4-7 참고)

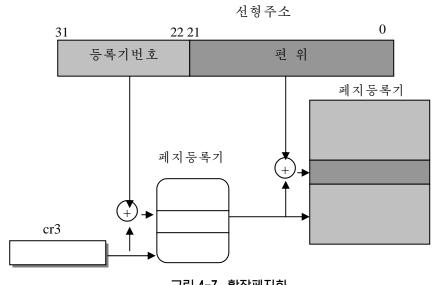



그림 4-7. 확장페지화

앞에서 본바와 같이 페지등록부입구의 Page Size 기발을 1로 설정하면 확장페지화가 가능해진다. 확장페지화를 사용하면 페지화유니트는 32bit선형주소를 다음과 같은 두 마당으로 구분한다.

등록부

웃자리10bit

편위

나머지 22bit

확장페지화에서 시용하는 페지등록부입구는 다음 사항을 제외하면 일반적인 페지화와 같다.

- Page Size기발을 1로 설정해야 한다.
- 20bit 물리주소 마당중 웃자리 10bit만 의미가 있다. 이것은 물리주소를 4MB 단위로 정렬하므로 주소의 아래자리 22bit는 0이 되기때문이다.

cr4처리기등록기의 PSE기발을 설정하면 확장페지화가 정규페지화와 공존할수 있다. 확장페지화는 크기가 아주 큰 련속된 선형주소를 이에 대응하는 물리주소로 그대로 변환할 때 사용한다. 이 경우 핵심부는 중간에 별도의 페지표를 구성할 필요가 없으므로 기억기를 아끼고 TLB입구를 보존할수 있다.

3) 하드웨어보호방책

페지화유니트와 토막화유니트는 서로 다른 보호방책을 사용한다. 80x86처리기에서는 토막마다 네 단계의 특권준위을 지정할수 있다. 그러나 페지와 페지표는 《정규페지

화》에서 언급한 User/Supervisor기발로 특권을 조종하기때문에 두가지 특권준위만 사용할수 있다. 이 기발이 0이면 CPL이 3보다 낮은 경우(즉 Linux에서 처리기가 핵심부방식에 있을 때)에만 해당 페지에 접근할수 있으며 이 기발이 1이면 해당 페지에 항상 접근할수 있다.

나아가서 토막에는 세 종류의 접근권한(읽기, 쓰기, 실행)을 지정할수 있지만 페지에는 읽기와 쓰기 두 종류의 접근권한만 설정할수 있다. 페지등록부입구나 페지표입구의 Read/Write기발이 0인 경우 해당 페지표나 페지는 읽기만 가능하고 기발이 1인 경우에는 읽기와 쓰기가 모두 가능하다.

페지화가 어떻게 동작하는가를 쉽게 리해하기 위해 다음 례를 살펴보자.

핵심부가 어떤 프로쎄스에 0x20000000부터 0x2003ffff까지 선형주소공간을 할당하였다고 하자. 이 주소공간은 정확히 64개의 페지로 이루어진다. 사실 이중 일부는 주기억기에 없을수도 있다. 여기서 관심을 둘 부분은 페지표입구의 나머지 마당이다.

먼저 프로쎄스에 할당한 선형주소의 웃자리 10bit부터 시작하자. 페지화유니트는 이것을 등록부마당으로 해석한다. 주소의 웃자리 10bit는 모두 2로 시작해서 나머지는 0이며 이 주소범위에 있는 모든 주소의 웃자리 10bit는 똑같은 값 즉 0x080, 10진수로는 128을 가진다.

따라서 모든 주소의 등록부마당은 프로쎄스에 할당된 페지등록부의 129번째 입구를 참조한다. 해당 입구에는 프로쎄스에 할당된 페지표의 물리주소가 있어야 한다.(그림 4-8참고) 프로쎄스에 다른 선형주소를 할당하지 않았다면 페지등록부의 남은 입구 1023개는 모두 0으로 채운다.

다음에 보지만 선형주소공간은 3GB까지 사용할수 있으나 사용자프로쎄스는 이 공간의 일부만 접근할수 있다.

중간10비트(즉 표 마당)의 값은 0에서 0x03f 10진수로는 0에서63까지의 범위에 해당한다. 따라서 페지표의 처음 64개 입구만 중요하고 나머지 입구 960개는 0으로 채운다.

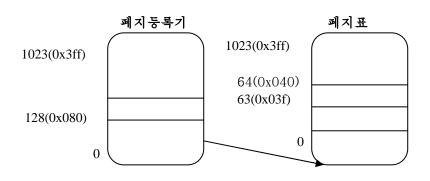



그림 4-8. 페지화의 례

프로쩨스가 선형주소 0x20021406에 있는 값을 읽으려고 하면 페지화유니트는 이 주소를 다음과 같이 처리한다.

1. 등록부마당 0x80은 페지등록부의 0x80번째 입구를 선택한다. 여기에는 프로쩨스가 사용하는 페지와 관련된 페지표의 위치가 들어 있다.

2.표 마당 0x21은 페지표의 0x21번째 입구를 선택한다. 여기에는 원하는 페지가 있는 페지틀의 위치가 들어 있다.

3.마지막으로 편위마당은 0x406이므로 페지틀의 0x406편위에 있는 값을 선택한다.

페지표의 0x21번째 입구의 Present기발이 0이라면 페지가 주기억기에 존재하지 않으므로 페지화유니트는 선형주소를 변환하는 과정에서 페지례외를 발생시킨다. 프로쩨스가 0x20000000에서 0x2003ffff범위밖에 있는 선형주소에 접근하는 경우에도 프로쩨스에 할당되지 않은 페지표는 0으로 채워지고 이것들의 Present기발은 모두 0이므로 마찬가지로 페지례외가 발생한다.

4) 3단계페지화

32bit극소형처리기는 2단계페지화(two-level paging)를 사용한다. 그러나 최근에 HP의 Alpha나 Intel의 Itanium, SUN의 UltraSPARC 같은 일부처리기에서 64bit구성방식를 도입하였다.

이 경우 2단계페지화가 더는 적합하지 않고 3단계페지화로 단계를 높여야 한다. 그리유를 생각해보자.

합리적이면서 가능한 큰 페지크기를 생각해보자.(빈번히 페지를 디스크에서 읽어들이고 써야 한다는것을 고려해야 한다.) 페지크기로 16kB를 선택해보자. 1kB는 주소 2^{10}이므로 16kB는 주소 2^{14}이다. 따라서 편위마당의 길이는 14bit가 된다. 그러면 남은 선형주소의 50bit를 표마당과 등록부마당으로 나누어야 한다. 이 두 마당을 각각 25bit로 정하면 각 프로쩨스의 페지등록부와 페지표는 2^{25}개 즉 3200만개가 넘는 입구를 포함하게 된다.

RAM이 아무리 눅어도 페지표를 저장하는데 이렇게 많은 기억기를 랑비할수는 없을것이다. 그래서 HP의 Alpha극소형처리기(시장에 처음으로 출하된 64bit CPU중 하나이다.)는 다음과 같은 해결책을 선택하였다.

- 페지틀의 크기는 8kB이다. 따라서 편위마당의 길이는 13bit이다.
- 주소에서 아래자리 43bit만 사용한다.(웃자리 21bit는 항상 0으로 설정한다.)
- 3단계 페지표를 사용하여 주소의 나머지 30bit를 10bit 세개로 나눈다. 따라서 페지표는 앞에서 본 2단계페지화방책을 사용할 때와 마찬가지로 입구 2^{10}=1024개를 포함한다.

후에 《Linux페지화》에서 보지만 Linux를 설계한 사람은 Alpha구성방식에서 착상하여 페지모형을 구성하기로 하였다.

물리주소확장페지화수법

처리기가 지원하는 RAM의 최대크기는 주소모선에 련결된 주소단자의 개수에 따라 결정된다. 80386부터 펜티움까지 조금 오래된 Intel처리기는 32bit크기의 물리주소를 사용하였다. 이런 체계에서는 리론적으로 RAM을 4GB까지 설치할수 있지만 실제로는 사용자방식프로쎄스에도 선형주소공간을 할당해야 하기때문에 핵심부는 1GB이상의 RAM에 직접접근할수 없다. 이점은 후에 《Linux페지화》에서 본다.

그렇지만 대형봉사기에서 실행하는 일부 응용프로그람은 1GB이상의 RAM을 필요로 하는데 이것은 최근 Intel이 32bit 80386구성방식에서 지원할수 있는 RAM의 크기를 늘이게 하였다.

Intel은 처리기의 주소단자개수를 32개에서부터 36개로 늘여서 이러한 요구에 대응하였다. 펜티움 pro부터 나온 모든 처리기는 이제 RAM을 2^{36}=64GB까지 다룰수 있다. 그렇지만 이렇게 늘어난 물리주소범위는 32bit선형주소를 36bit물리주소로 바꾸는 새로운 수법을 적용해야만 사용할수 있다.

Intel은 펜티움pro처리기부터 물리주소확장(Physical Address Extension)이라는 수법을 도입하였다. 펜티움Ⅲ처리기부터 《 페지크기확장(PSE036, Page Size Extension)》이라는 또 다른 수법이 등장했는데 Linux에서는 이것을 사용하지 않으므로 여기서도 취급하지 않는다.

cr4조종등록기의 물리주소확장(PAE)기발을 설정하면 PAE가 활성화된다. cr4페지등록부입구의 페지크기(PS)기발을 설정하면 큰 페지크기(PAE를사용하는 경우2MB)를 사용할수 있다.

Intel은 PAE를 지원하기 위해 다음과 같이 페지화수법을 수정하였다.

- RAM 64GB를 개별페지를 2^{24}개로 나누고 페지표입구의 물리주소마당길이를 20bit에서 24bit로 늘였다. PAE페지표의 매 입구마다 12bit의 기발이 필요하고 물리주소로 24bit가 필요하므로 합쳐서 36bit가 필요하다. 따라서 페지표입구의 크기가 32bit에서 64bit로 즉 두배로 늘어났다. 그 결과 4kB의 PAGE페지표에는 1024개가 아닌 512개의 입구가 들어간다.
- 페지등록부지적자 표(PDPT:Page Directory Pointer Table)이라는 새로운 단계의 페지표를 추가하였다. 여기에는 64bit 입구 4개가 들어간다.
- cr3조종등록기에는 27bit크기의 페지등록기지적자표(PDPT)의 시작주소마당이 있다. PDPT는 처음 4GB RAM에 저장되고 32B(2^5)의 배수로 정렬되므로 이 표의 시작주소를 나타내는데는 27bit면 충분하다.
- 선형주소를 4kB 페지로 배치할 때(페지등록부입구의 PS기발은 0) 선형주소의 32bit를 다음과 같이 해석한다.

cr3

PDPT를 가리킨다.

31-30bit

PDPT에 들어있는 입구 4개중 하나를 가리킨다.

29-21bit

페지등록부에 있는 입구 512중 하나를 가리킨다.

20-12bit

페지표에 있는 입구 512개중 하나를 가리킨다.

11-0bit

4kB 페지에서의 편위

- 선형주소를2MB 페지로 배치할 때(페지등록부입구의 PS 기발은 1) 선형주소의 32bit를 다음과 같이 해석한다.

cr3

PDPT를 가리킨다.

31-30bit

PDPT에 들어있는 입구 4개중 하나를 가리킨다.

29-21bit

페지등록부에 있는 입구 512개중 하나를 가리킨다.

20-0bit

2MB페지에서의 편위

일단 cr3을 설정하면 4GB까지의 RAM을 접근할수 있다. RAM을 이보다 크게 사용하려면 cr3에 새로운 값을 지정하거나 PDPT의 내용을 수정해야 한다. 그러나 PAE의 큰 문제는 여전히 선형주소가 32bit크기라는 점이다. 즉 프로그람작성자는 RAM의 다른 령역을 접근하는데도 같은 선형주소를 재사용해야 한다. 후에 《RAM 크기가 4096MB이상일 때 최종핵심부페지표》에서 PAE를 사용할 때 Linux가 페지표를 어떻게 초기화하는가를 간단히 본다.

5) 하드웨어캐쉬

현재 극소형처리기의 박자(clock)속도는 몇 GHz에 이르지만 DRAM(Dynamic RAM)소자에 접근하는데는 수백박자가 요구된다. 따라서 피연산자를 RAM에서 가져오거나 결과를 RAM에 저장할 때 CPU는 상당히 오래동안 멈추어 있어야 한다.

이러한 CPU와 RAM의 속도차이를 줄이려고 하드웨어캐쉬기억기(hardware cache memory)가 등장하였다. 캐쉬는 많이 알고있는 국부성원리(locality principle)에 기초한다. 이 원리는 프로그람코드와 자료구조 모두에 해당하는것이며 프로그람은 순환구조를 가지고 서로 련관된 자료는 련속된 배렬에 몰려있기때문에 가장 최근에 사용한 주소와 가까운곳에 있는 주소를 가까운 시간에 사용할 가능성이 가장 높다는것이다. 따라서 작고 빠른 기억기에 가장 최근에 사용한 코드와 자료를 넣어두는것이

합리적이다. 이런 목적으로 80x86구성방식에 선로(line)라는 새로운 단위가 등장하였다. 이것은 소자에 들어있고 캐쉬를 구성하는데 사용하는 빠른 SRAM(정적RAM: Static RAM)과 느린 DRAM사이에서 묶음방식(burst mode)으로 전송되는 련속된 바이트 몇개로 이루어진다.

캐쉬는 선로의 부분집합으로 나누어진다. 캐쉬를 구성할 때 극단적으로 직접배치(direct mapped)하는 방법이 있다. 이때에는 주기억기에 있는 선로를 캐쉬에서 항상 완전히 똑같은 위치에 저장한다. 이와 정반대로 캐쉬를 완전조합(fully associative)으로 구성하는 방법이 있다. 이 경우 기억기에 있는 모든 선로을 캐쉬의 어느 위치에나 저장할수 있다.

그러나 대부분의 캐쉬는 N-방향집합 조합(N-way set associative)으로 구성한다. 여기서는 주기억기의 어떤 선로이든 캐쉬의 N선로중 하나에 저장할수 있다. 례를 들어 기억기의 한 선로는 2-방향 집합조합 캐쉬에 있는 2개의 다른 선로에 저장할수 있다.

그림 4-9에서 볼수 있는것처럼 캐쉬유니트는 페지화유니트과 주기억기사이에 위치한다. 캐쉬유니트에는 하드웨어캐쉬기억기(hardware cache memory)와 캐쉬조종기(cache controller)가 들어있다.

캐쉬기억기는 실제로 기억기선로를 저장하고 캐쉬조종기는 매 선로마다 입구 하나씩 입구배렬을 포함한다. 매 입구에는 꼬리표와 캐쉬선로의 상태를 나타내는 몇가지 기발이 있다. 이 꼬리표는 캐쉬조종기가 선로에 배치된 현재 기억기의 위치를 인식할수 있게 해주는 여러 비트로 이루어진다. 기억기 물리주소비트는 보통 세 그룹으로 나누어지는데 이중 가장 웃자리비트는 꼬리표에, 가운데 비트는 캐쉬조종기집합색인에, 마지막 비트는 선로의 편위에 해당한다.

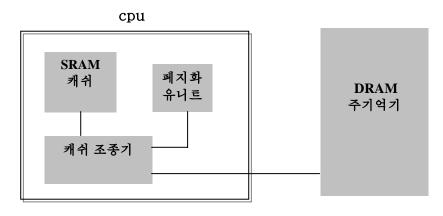



그림 4-9. 처리기하드웨어캐쉬

RAM의 기억기요소에 접근할 때 CPU는 물리주소에서 집합의 색인을 뽑아내여 이 집합에 들어있는 모든 선로의 꼬리표와 물리주소의 웃자리비트를 서로 비교한다. 그래서 주소의 웃자리비트와 꼬리표가 같은 선로을 발견하면 CPU는 캐쉬명중(cache hit)을

하고 그렇지 않으면 캐쉬실패(cache miss)가 발생한다.

캐쉬명중이 일어나면 캐쉬조종기는 접근류형에 따라 다르게 동작한다. 읽기동작이면 조종기는 캐쉬선로의 자료를 선택하여 이것을 CPU등록기에 전송한다. 여기서는 RAM에 접근하지 않기때문에 캐쉬체계를 개발한 목적 그대로 CPU는 시간을 절약할수 있다. 쓰기동작인 경우 조종기는 쓰기를 실현하는 두가지 기본방책인 직접쓰기(write-through)와 지연쓰기(write-back)중 하나를 실현할수 있다. 직접쓰기를 하는 경우 조종기는 항상 RAM과 캐쉬선로에 동시에 쓰기때문에 실질적으로 쓰기동작에 대해서는 캐쉬를 끄는 효과를 가져온다. 지연쓰기는 직접쓰기보다 즉시적인 효률성을 제공하는데 이때에는 캐쉬선로만 갱신하고 RAM내용은 바꾸지 않는다. 응당 후에 쓰기를 한 후 언젠가는 RAM의 내용을 갱신해야 한다. 캐쉬조종기는 CPU가 캐쉬입구를 흘리기(flush)해야 하는 명령을 실행하거나 FLUSH하드웨어신호가 발생한 경우(보통 캐쉬실패가 일어난 후)에만 캐쉬선로의 내용을 RAM에 기록한다.

캐쉬실패가 발생한 경우 필요하면 캐쉬선로를 RAM에 저장하고 정확한 선로를 RAM에서 캐쉬입구로 가져온다.

다중처리기체계에는 모든 처리기마다 별도의 하드웨어캐쉬가 있으므로 매 캐쉬의 내용을 동기화하기 위한 별도의 하드웨어회로가 필요하다. 그림 4-10에서 보는것처럼 각 CPU는 자기만의 국부하드웨어캐쉬를 가진다. 여기서는 캐쉬를 갱신하는것이 시간을 더 많이 필요로 하는 작업이 된다. CPU는 자기의 하드웨어캐쉬내용을 바꿀 때마다 다른 하드웨어캐쉬에 같은 자료가 들어 있는가를 반드시 확인해야 하며 있다면 다른 CPU에 캐쉬를 정확한 값으로 갱신하라고 알려주어야 한다. 이런 작업을 종종 캐쉬조사(cache snooping)라고 한다. 다행히도 이런 작업들은 하드웨어준위에서 이루어지며 핵심부는 전혀 신경쓰지 않아도 된다.

캐쉬기술은 빠르게 발전하고있다. 례를 들어 처음 나온 펜티움모형은 소자에 《L1-캐쉬》라는 캐쉬 하나만 포함하였다. 이 보다 후에 나온 모형은 소자에 이보다 더 크지만 느린 《L2-캐쉬》라는 다른 캐쉬를 포함한다. 두 캐쉬의 내용을 일치시키는것 역시 하드웨어준위에서 실현된다. Linux는 이런 하드웨어의 세세한 내용은 무시하고 캐쉬가 하나만 있다고 여긴다.

cr0처리기등록기의 CD기발은 캐쉬회로를 사용하거나 금지하는데 사용한다. 똑같은 등록기에 있는 NW기발은 캐쉬에 직접쓰기방책을 사용하겠는지 지연쓰기방책을 사용하겠는지를 지정한다.

펜티움처리기에 있는 캐쉬의 또 다른 흥미있는 특징은 조작체계가 매 페지틀마다 다른 캐쉬관리방책을 사용할수 있게 하는것이다. 이를 위해 각 페지등록부와 페지표입구마다 기발이 두개 있다. 페지캐쉬끄기(PCD:Page Cache Disable)는 해당 페지틀에 있는 자료에 접근할 때 캐쉬를 사용하겠는가 하는 여부를 정한다. 페지직접쓰기(PWT:Page Write-Through)는 해당 페지틀에 자료를 쓸 때 직접쓰기방책을 사용하겠는가

지연쓰기방책을 사용하겠는가를 지정한다. Linux는 모든 페지등록부와 페지표입구의 PCD와 PWT기발을 지운다. 따라서 모든 페지틀에서 캐쉬를 사용하고 쓰기를 할 때에는 항상 지연쓰기방책을 채택한다.

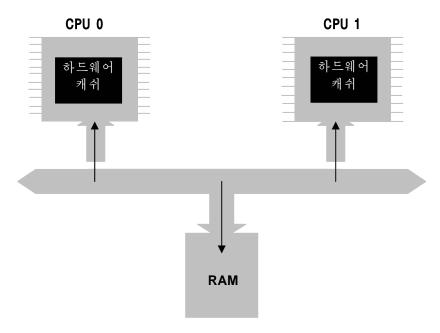



그림 4-10. 처리기가 두개인 체계에서의 캐쉬

6) 변환참조완충기

범용하드웨어캐쉬외에도 80x86처리기에는 《변환참조완충기(TLB:translation lookaside buffer)》라는 다른 종류의 캐쉬가 있어서 선형주소를 변환하는 속도를 높여준다. 어떤 선형주소를 처음 사용하는 경우 느린 RAM에 있는 페지표에 접근해서 해당 물리주소를 계산한다. 그러면 변환된 물리주소를 TLB입구에 저장함으로써 앞으로 똑같은 선형주소를 참조하는 경우 빠르게 변환할수 있게 한다.

다중처리기체계에서는 매 CPU마다 자기만의 TLB가 있는데 이것을 《국부TLB(local TLB)》라고 한다. L1캐쉬와는 반대로 TLB입구들을 같은 곳에 대한 입구를 서로 련결시킬 필요가 없는데 여러 CPU에서 실행하는 프로쎄스들이 같은 선형주소를 사용하더라도 이것이 다른 물리주소와 련결되여있을수 있기때문이다.

CPU에 있는 cr3조종등록기가 바뀌면 하드웨어는 자동으로 국부TLB에 있는 모든 입구를 무효화한다.

5. Linux페지화

《3단계페지화》에서 설명한것처럼 Linux는 3단계페지화모형을 선택하여 64bit구성방식에도 적합하다. Linux는 다음 세 종류의 페지표를 정의하며 그림 4-11은 이러한

Linux의 페지화모형을 보여준다.

- 페지대역등록부(Page Global Directory)
- 페지중간등록부(Page Middle Directory)
- 페지표(Page Table)

페지대역등록부는 여러 페지중간등록부의 주소를 포함하고 페지중간등록부는 다시 여러 페지표의 주소를 포함한다. 매 페지표입구는 페지틀을 가리킨다. 따라서 선형주소는 4부분으로 나누어진다. 그림 4-11에서는 매 부분의 비트수를 표시하지 않았는데 이것은 각 부분의 크기는 콤퓨터구성방식에 따라 다르기때문이다.

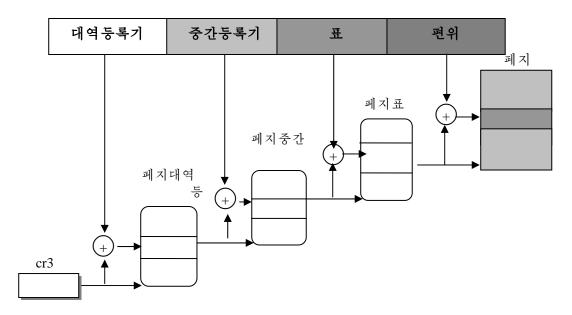



그림 4-11. 리눅스페지화 모형

Linux는 프로쎄스를 다룰 때 페지화에 많이 의존한다. Linux는 선형주소를 물리주소로 자동변환하여 다음과 같은 설계목적을 달성한다.

- 매 프로쎄스에 서로 다른 물리주소공간을 할당해서 주소지정오유를 효과적으로 차단한다.
- 페지(자료그룹)를 페지틀(주기억기에 있는 물리주소)과 구별한다. 이것은 똑같은 페지를 페지틀에 저장하였다가 디스크에 저장하고 후에 다른 페지틀로 적재할수 있게 하며 가상기억기수법의 기본적인 요소이다.

앞에서 보았지만 매 프로쎄스는 자기만의 페지대역등록부와 일련의 페지표를 가진다. 프로쎄스절환이 일어나면 Linux는 cr3조종등록기의 값을 앞서 실행중이던 프로쎄스의 서술자에 저장하고 다음에 실행할 프로쎄스의 서술자에 저장된 값을 가져와 cr3을 설정한다. 따라서 CPU가 새로운 프로쎄스의 실행을 재개할 때 페지화유니트는 정확한 페

지표집합을 참조하게 된다.

두 종류의 페지표만 있는 펜티움에 3단계페지화모형을 적용하면 어떻게 되는가?

Linux는 페지중간등록부마당의 길이가 0bit라고 표시하여 페지중간등록부를 본질적으로 제거한다. 그러나 일련의 지적자에서 페지중간등록부의 위치는 그대로 남아서 똑같은 코드로 32bit와 64bit구성방식 모두에서 동작하게 한다. 핵심부는 페지중간등록부에 있는 입구의 수를 1로 설정하고 이 입구 하나를 페지대역등록부에 있는 적합한 입구로 배치하여 페지중간등록부의 위치를 유지한다.

그러나 Linux가 펜티움 pro나 그 후에 나온 처리기에서 지원하는 물리주소확장(PAE)수법을 사용하는 경우 Linux의 페지대역등록부는 80x86의 페지등록부에, Linux의 페지표는 80x86의 페지표에 각각 해당한다.

론리주소를 선형주소로 배치하는것은 기계적인 작업이 되였지만 여전히 복잡한 작업으로 되고있다. 따라서 다음 부분에서는 핵심부가 주소를 찾고 표를 관리하는데 필요한 정보를 가져오는 함수와 마크로의 목록을 서술한다. 이 함수의 코드는 대부분 한행이나 두행정도이다. 뒤에서 이 함수와 마크로를 자주 다루기때문에 이것들의 역할에 대해서 취급하려 한다.

1) 선형주소마당

다음 마크로를 리용하여 간단하게 페지표를 관리할수 있다.

PAGE_SHIFT

편위마당의 비트수를 지정한다. 80x86처리기에서 이 값은 12이다. 편위마당의 길이는 한 페지에 있는 모든 주소가 들어갈수 있게 지정해야 한다. 80x86체계에서 한 페지의 크기는 2^{12}, 즉 4096B이다. 따라서 PAGE_SHIFT값은 전체 페지크기에 2를 밑수으로 하는 로그를 취한 값인 12이다. PAGE_SIZE는 이 마크로를 사용하여 페지의 크기를 돌려준다. 마지막으로 PAGE_MASK마크로를 0xfffff000으로 정의한다. 이 마크로는 편위마당의 모든 비트를 마스크하는데 사용한다.

PMD_SHIFT

선형주소의 페지편위와 페지표마당의 전체 길이 즉 한 페지중간등록부입구가 배치할수 있는 령역의 크기에 로그를 취한 값이다. PMD_SIZE 마크로는 페지중간등록부의 입구 하나 즉 한 페지표에서 배치하는 령역의 크기를 계산한다. PMD_MASK마크로는편위와 표마당을 모두 마스크한다.

PAE를 사용하지 않을 때 PMD_SHIFT의 값은 22이고(편위마당에서 12, 표마당에서 10), PMD_SIZE는 2^{22} 즉 4MB이며 PMD_MASK는 0xffc00000이다. PAE를 사용할 때에는 PMD_SHIFT의 값이 21이고(편위마당에서 12, 표마당에서 9), PMD_SIZE는 2^{21} 즉 2MB이며 PMD_MASK는 0xffc00000이다.

PGDIR_SHIFT

한 페지대역등록부입구가 배치할수 있는 령역의 크기에 로그를 취한값이다.

PGDIR_SIZE마크로는 페지대역등록부의 입구 하나에서 배치하는 령역의 크기를 계산한다. PGDIR_MASK마크로는 편위와 표, 중간등록부마당을 모두 마스크한다.

PAE를 사용하지 않을 때 PGDIR_SHIFT의 값은 22이고(PMD_SHIFT의 값과 같다.) PGDIR_SIZE 는 2^{22} 즉 4MB이며 PGDIR_MASK는 0xffc00000이다. PAE를 사용할 때에는 PGDIR_SHIFT의 값이 21이고(편위 마당에서 12, 표 마당에서 9, 중간등록부마당에서 9), PGDIR_SIZE는 2^{30} 즉 1GB이며 PGDIR_MASK는 0xc0000000이다.

PTRS_PER_PTE, PTRS_PER_PMD, PTRS_PER_PGD

페지표와 페지중간등록부, 페지대역등록부에 들어가는 입구의 수를 계산한다. PAE를 사용하지 않을 때 각각의 값은 1024, 1, 1024이며 PAE를 사용할 때에는 각각 512, 512, 4이다.

2) 페지표다루기

pte_t와 pmd_t, pgd_t는 각각 페지표와 페지중간등록부, 페지대역등록부입구의 형을 나타낸다. 이것들은 PAE를 사용불가능할 때에는 32bit자료형이고 가능할 때에는 64bit자료형이다. pgprot_t는 한 입구와 관련된 보호비트(protection bit)를 나타내는 또 다른 32bit형이다.

형을 변환하는 4개 마크로 __pte(), __pmd(), __pgd(), __pgprot()는 unsigend long형를 필요한 형으로 변환한다. 형을 변환하는 다른 4개 마크로 pte_val(), pmd_val(), pgd_val(), pgprot_val()은 반대로 특정형에서 unsigend long형으로 변환한다.

핵심부는 또한 페지표입구를 읽고 수정하는 여러 마크로와 함수를 제공한다.

- pte_none(), pmd_none(), pgd_none() 마크로는 해당 입구값이 0이면 1, 그렇지 않으면 0을 반환한다.
- pte_ present(), pmd_present(), pgd_present()마크로는 해당 입구 Present기발이 1, 즉 해당 페지나 페지표가 주기억기에 적재되여있으면 1을 반환한다.
- pte_clear(), pmd_clear(), pgd_clear()마크로는 해당 페지표의 입구를 지운다. 따라서 프로쎄스는 해당 페지표입구가 배치하고있던 선형주소를 더는 사용할수 없게 된다.

pmd_bad()와pgd_bad()는 함수가 입력파라메터로 전달받은 페지대역등록부와 페지중간등록부의 입구를 검사하는데 사용한다. 각 마크로는 입구가 잘못된 페지표를 가리키고있으면 즉 다음 조건중 하나라도 해당하면 1을 되돌린다.

- 페지가 주기억기에 존재하지 않는다.(Present기발이 0이다.)
- 페지에 읽기접근만 할수 있다.(Read/Write기발이 0이다.)
- Accessed나 Dirty가 0이다.(Linux는 존재하는 모든 페지표에 있는 이 기발을 항상 1로 설정한다.)

pte_bad()라는 마크로는 없는데 그 리유는 페지표입구가 주기억기에 존재하지 않거

나 쓸수 없거나 접근할수 없는 페지를 가리키는것 모두 합법적이기때문이다. 대신 페지표입구에 들어있는 기발의 현재값을 알아내는 여러 함수를 제공한다.

pte_read()

user/supervisor기발값을 되돌린다.(페지를 사용자방식에서 접근할수 있는가를 나타낸다.)

pte_write()

read/write기발이 설정되여있으면 1을 되돌린다.(페지에 쓰기가능한가를 나타낸다.)

pte_exec()

user/supervisor기발값을 되돌린다.(사용자방식에서 페지에 접근할수 있는가를 나타낸다.)

80x86처리기에서는 페지에 있는 코드실행을 금지할수 있는 방법이 없다.

pte_dirty( )

dirty기발값을 되돌린다.(페지에 있는 내용이 바뀌였는가를 나타낸다.)

pte_young()

accessed 기발값을 되돌린다.(페지에 접근했는가를 나타낸다.)

페지표입구에 있는 기발값을 설정하는 다른 함수그룹도 있다.

pte_wrprotect()

read/write기발을 0으로 설정한다.

pte_rdprotect()와 pte_exprotect()

User/Supervisor기발을 0으로 설정한다.

pte_mkwrite()

Read/Write기발을 1로 설정한다.

pte_mkread()와 pte_mkexec()

User/Supervisor기발을 1로 설정한다.

pte_mkdirty()와 pte_mkclean()

Dirty기발을 각각 1과 0으로 설정해서 페지가 바뀌였거나 바뀌지 않았다고 표시한다.

pte_mkyoung()과 pte_mkold()

Access기발을 각각 1과 0으로 설정해서 페지에 누군가 접근했거나(young) 접근하지 않았다고(old) 표시한다.

pte_ modify(p, v)

페지표입구 p의 모든 접근권한을 v에 지정한 값으로 설정한다.

set_pte와 set_pmd, set_pgd

지정한 값을 각각 페지표, 페지중간등록부, 페지대역등록부입구에 기록한다.

ptep_set_wrprotect()와 ptep_mkdirty()함수는 pte_wrprotect()와 pte_mkdirty()함수와 비슷하지만 페지표입구에 대한 지적자를 파라메터로 받는다는 점이 다르다.

ptep_test_and_clear_dirty()와 ptep_test_and_clear_young()함수는 pte_mkclean()과 pte_mkold()함수와 비슷하지만 마찬가지로 페지표에 대한 지적자를 파라메터로 받으며 기발에 있던 이전값을 돌려준다는 점이 다르다.

다음은 페지주소와 여러 보호기발을 조합하여 페지입구를 만들거나 이와 반대로 페지표입구에서 페지주소를 뽑아내는 마크로이다.

mk_pte

선형주소와 여러 접근권한을 조합해서 페지표입구를 만든다.

mk_pte_phys

물리주소와 페지접근권한을 조합해서 페지표입구를 만든다.

pte_page

페지표입구가 참조하는 페지틀의 서술자의 주소를 되돌린다.

pmd_page( )

페지중간등록부입구에서 페지표의 선형주소를 가져와 되돌린다.

pgd_offset(p,a)

이 마크로는 기억기서술자 p와 선형주소 a를 파라메터로 받아 페지대역서술자에서 주소 a에 해당하는 입구의 주소를 반환한다. 페지대역등록부는 기억기서술자 p에 있는 지적자를 리용하여 찾을수 있다. pgd_offset_k()마크로는 이와 비슷하지만 주핵심부페지표를 참조한다는 차이가 있다.

pmd_offset(p,a)

이 마크로는 페지대역등록부의 입구 p와 선형주소 a를 파라메터로 받아 p가 참조하는 페지중간등록부에서 주소 a에 해당하는 입구의 주소를 반환한다.

지금까지 살펴본 긴 목록에서 마지막에 나온 함수들은 페지표입구를 쉽게 만들고 지우기 위하여 도입한것이다.

2단계페지화를 사용할 때(PAE는 사용하지 않고) 페지중간등록부입구를 만들고 지우는 일은 매우 간단하다. 앞에서 설명한바와 같이 페지중간등록부는 자기에 종속된 페지표를 가리키는 입구 하나만을 가진다. 따라서 페지표중간등록부입구는 페지대역등록부에 있는 입구이기도 하다. 그러나 페지표를 다룰 때 입구를 만드는 작업은 해당 입구를 포함해야 할 페지표가 존재하지 않을수도 있기때문에 조금 더 복잡하다. 이런 경우 새로운 페지틀을 할당하여 그것을 0으로 채운 다음 입구를 추가해야 한다.

PAE를 사용하는 경우 핵심부는 3단계페지화를 사용한다. 핵심부는 새 페지대역등록부를 만들 때 이에 따르는 페지중간등록부 4개를 함께 할당한다. 이것들은 부모격인 페지대역등록부가 없어질 때에만 해제된다.

《페지틀관리》에서 보지만 페지틀을 할당하고 해제하는 일은 시간이 많이 소요되는 작업이다. 따라서 핵심부는 페지표를 없앨 때 기억기를 해제하는 대신 해당 페지틀을 적당한 기억기캐쉬에 추가한다.

pgd_alloc(m)

get_zeroed_page()함수를 호출하여 새로운 페지대역등록부를 할당한다. PAE를 사용하는 경우 이 함수는 4개의 페지중간등록부도 함께 할당한다. 80x86구성방식에서는 파라메터 m을(기억기서술자의 주소) 무시한다.

pmd_alloc(m, p, a)

3단계페지화체계에서 선형주소 a에 해당하는 새로운 페지중간등록부를 할당할 수 있도록 정의한 함수이다. PAE를 사용하지 않을 때에는 이 함수는 입력인자 p, 즉 페지대역등록부에 있는 입구의 주소를 그대로 반환한다. PAE를 사용하는 경우 이 함수는 페지대역등록부를 만들었을 때 할당한 페지중간등록부의 주소를 반환한다. 파라메터 m은 무시한다.

pte_alloc_kernel(m, p, a)

페지중간등록부 입구 p의 주소와 선형주소 a를 인자로 받아서 a에 해당하는 페지표입구의 주소를 반환한다. 페지중간등록부 입구가 비였으면(null) 이 함수는 새로 페지표를 할당한다. 이것을 위해 페지틀을 할당할 때에는 pte_alloc_one_kernel()함수를 호출한다. 새로 페지표를 할당하고 나면 a에 해당하는 입구를 초기화하고 User/Supervisor기발을 1로 설정한다. 파라메터 m은 무시한다.

pte_free()와 pgd_free()

페지표를 해제한다. 페지중간등록부는 부모격인 페지대역등록부와 함께 할당하고 해제하므로 pmd_free()함수는 아무 일도 하지 않는다.

clear_page_tables()

free_one_pgd()함수를 여러번 호출해서 프로쎄스의 페지표의 내용을 지운다.

3) 예약된 페지틀

핵심부코드와 자료구조는 일련의 예약된 페지틀(reserved page frame)에 저장한다. 이 페지틀에 들어있는 페지는 절대로 동적으로 할당하거나 디스크로 교환하지 않는다. 일반적으로 Linux핵심부를 RAM의 물리주소 0x00100000 즉 1MB령역부터 설치한다. 필요한 페지틀의 전체개수는 핵심부를 어떻게 설정하였는가에 따라 달라지지만 일반설정인 경우 핵심부는 RAM을 2MB이하로 사용한다.

왜 핵심부를 기억기의 처음 1MB령역으로 적재하지 않는가?

이와 관련하여 PC구성방식에서 다음과 같은것을 고려해야 한다.

- 페지틀 0은 BIOS가 전원투입시 자기진단(POST:Power-On Self-Test)과정에서 인식한 체계하드웨어설정을 저장하는데 사용한다. 또한 휴대용콤퓨터에 있는 많은 BIOS는 체계를 초기화한 후에도 이 페지에 자료를 기록하기도 한다.
- 물리주소 0x000a0000에서 0x000fffff까지는 BIOS함수와 ISA화면표시카드의 내부기억기를 배치하는 용도로 예약되여있다. 이 령역은 모든 IBM호환PC에 있는 640kB부터 1MB까지의 빈 령역(hole)으로 잘 알려져있다. 물리주소는 존재하지만 다

른 용도로 예약되여있어서 조작체계는 해당 페지들을 사용할수 없다.

- 일부특정콤퓨터모형에서는 처음 IBM안에 있는 페지들을 추가로 예약하기도 한다. 례를 들어 IBM Think Pad는 0xa0페지들을 0x9f페지들에로 배치한다.

체계기동과정의 초기단계에서 핵심부는 BIOS에 문의하여 물리기억기의 크기를 알아낸다. 최근 콤퓨터에서 핵심부는 물리주소범위와 해당 기억기종류의 목록을 만드는 BIOS함수도 호출한다. 후에 핵심부는 setup_memory_region()함수를 실행한다. 이 함수는 표 4-1에서 보는것처럼 물리주소령역의 목록을 만든다. BIOS에서 목록을 제공하면 핵심부는 이것을 기반으로 목록을 작성하고 그렇지 않으면 전통적인 기본설정에 따라 목록을 작성한다. 0x9f(LOWMEMSIZE)부터 0x100(HIGH_MEMORY)사이의 번호를 가진 모든 페지들은 예약된것으로 표시한다.

표 4-1. BIOS에서 제공하는 물리주소 배렬의 례

시작	끝	종류
0x00000000	0x0009ffff	사용가능
0x000f0000	0x000fffff	예약됨
0x00100000	0x07feffff	사용가능
0x07ff0000	0x07ff2fff	ACPI자료
0x07ff3000	0x07ffffff	ACPI NVS
0xffff0000	0xffffffff	예약됨

표 4-1은 128MB의 RAM을 가지고있는 콤퓨터의 전형적인 기억기구성이다. BIOS는 POST단계에서 물리주소 0x07ff0000에서 0x07ff2fff사이의 령역에 체계의 하드웨어장치에 관한 정보를 저장한다. 핵심부는 초기화단계에서 이 정보를 적당한 핵심부자료구조로 복사한 후 이 페지들을 사용가능하다고 여긴다. 반대로 물리주소 0x07ff3000에서 0x07ffffff사이의 령역은 하드웨어장치에 있는 ROM소자로 배치된다.

0xffff0000에서 시작하는 물리주소령역은 하드웨어가 BIOS의 ROM소자를 배치하는데 사용하므로 예약된 령역으로 표시된다. BIOS는 어떤 물리주소범위에 대해서는 아무런 정보를 제공하지 않을수도 있다.(표에서는 0x000a0000에서 0x000effff까지가 비여있다.) 안전하게 하기 위해 Linux는 이런 범위를 사용할수 없는 령역이라고 여긴다.

Linux는 핵심부를 비련속적인 페지들로 적재하는것을 피하려고 RAM의 처음 1MB를 건너뛴다. PC구성방식에서 특별히 예약하지 않은 페지들은 Linux가 동적으로 할당한 페지를 저장하는데 사용할것이다.

그림 4-12에서는 Linux에서 RAM의 처음 2MB령역을 어떻게 채우는가를 보여준다. 여기서는 핵심부가 1MB보다 적은 RAM을 필요로 한다고 가정한다.

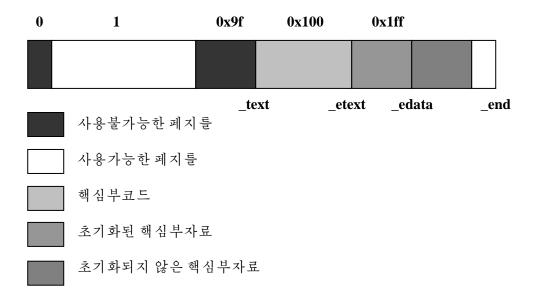



그림 4-12. 리눅스에서 처음 512개(2MB)의 페지들

물리주소 0x00100000에 해당하는 _text라는 기호는 핵심부코드의 첫번째 바이트의 주소를 나타낸다. 비슷하게 핵심부코드의 끝을 _etext로 나타낸다. 핵심부자료는 초기화된(initialized) 자료와 초기화되지 않은(uninitialized) 자료로 나누어진다. 초기화된 자료는 _etext바로 다음부터 시작하며 _edata에서 끝난다.

그 다음에 초기화되지않은 자료가 나오고 _edata에서 끝난다.

그림에 나오는 기호는 Linux원천코드에는 정의되여있지 않으며 핵심부를 콤파일하는 도중에 만들어진다.

○ **프로쎄스페지표**

프로쎄스의 선형주소공간은 두 부분으로 나누어진다.

- 0x00000000에서 0xbfffffff까지의 선형주소는 프로쎄스가 사용자방식에 있든 핵심부방식에 있든 항상 접근할수 있다.
- 0xc0000000에서 0xffffffff까지의 선형주소는 프로쎄스가 핵심부방식에 있을 때에만 접근할수 있다.

프로쎄스가 사용자방식에서 동작중일 때 프로쎄스는 0xc0000000보다 작은 선형주소를 만들어내고 핵심부방식에서 동작중일 때에는 핵심부코드를 실행하면서 0xc0000000이상의 선형주소를 만들어낸다. 그렇지만 때때로 핵심부는 자료를 가져오거나 저장하기 위해서 사용자방식의 선형주소공간에 접근해야만 한다.

PAGE_OFFSET마크로는 0xc0000000값을 가진다. 이것은 프로쎄스의 선형주소공

간에서 핵심부가 위치하는 편위이다. 이 책에서는 자주 이 마크로대신 0xc0000000값을 직접 사용한다.

0xc0000000보다 아래에 있는 선형주소를 배치하는 페지대역등록부의 처음 입구들(PAE를 사용하지 않을 때 입구 768개)의 내용은 프로쎄스마다 다르다. 반면에 나머지 입구들은 모든 프로쎄스에서 똑같아야 하며 핵심부주페지대역등록부에 있는 해당 입구와도 같다. 이 기호들의 선형주소는 핵심부콤파일직후에 만들어지는 System.map파일에서 찾을수 있다.

2.2핵심부에서는 1GB보다 크지만 2GB보다 작은 RAM을 사용하기 위해서 PAGE_OFFSET을 0xc0000000이 아닌 다른값으로 바꿀수 있었다. 하지만 2/4이상에 들어와서 4GB이상의 기억기를 지원하는 highmem을 쓰면서 이 값이 0xc0000000으로 고정되였다.

○ **핵심부페지표**

핵심부는 자기가 사용하기 위한 페지표도 관리하며 이것을 주핵심부페지대역등록부(master kernel Page Global Directofy)라고 한다. 체계를 초기화한 후에는 어떤 프로쎄스나 핵심부스레드도 이 페지표를 직접 사용하지 않는다. 그대신 주핵심부페지대역등록부의 웃자리에 있는 입구들은 체계에 있는 모든 정규프로쎄스의 페지대역등록부에 있는 해당 입구들의 참조모형이다.

주핵심부페지대역등록부가 바뀔 때 이것을 어떻게 체계에 있는 프로쎄스가 사용하는 페지대역등록부로 전달하는가는 《불련속적인 기억기령역 접근처리》에서 설명한다.

이제 핵심부가 자기의 페지표를 초기화하는 법을 살펴보자. 이것은 두 단계를 거친다. 핵심부사본을 기억기에 적재한 직후 CPU는 여전히 실방식(real mode)에서 동작하며 페지화는 사용하지 않는다.

첫번째 단계로 핵심부를 RAM에 넣는데 충분한 8MB크기의 제한된 주소공간을 만든다.

두번째 단계로 핵심부는 모든 RAM을 활용하여 페지표를 정확히 만든다. 이제 이 단계가 어떻게 진행되는지 보자.

· **림시핵심부페지표**

림시페지대역등록부는 핵심부를 콤파일할 때 정적으로 초기화하며 페지표는 arch/i386/kernel/head.S에 있는 startup_32()기호언어함수에서 초기화한다. 페지중간등록부는 페지대역등록부입구와 동일하므로 더는 언급하지 않겠다.

페지대역등록부는 swapper_pg_dir변수에 들어있고 RAM의 처음 8MB만 포함하는 페지표는 pg0과 pg1변수에 들어 있다.

페지화 첫째 단계에서는 이 8MB를 실제방식과 보호방식 모두에서 쉽게 접근할수 있게 한다. 그러기 위해 핵심부는 0x00000000에서 0x007fffff까지의 선형주소와 0xc0000000에서 0xc07fffff까지의 선형주소를 모두 0x00000000에서 0x007fffff까지의 물리주소로 배치한다. 다시말하여 초기화의 첫째 단계동안 핵심부는 처음 8MB의

RAM을 접근할 물리주소와 똑같은 선형주소를 사용할수도 있고 0xc0000000부터 8MB 범위에 있는 선형주소를 사용할수도 있다.

핵심부는 swapper_pg_dir의 0, 1, 0x300(십진수 768), 0x301(십진수 769)번째 입구를 제외한 나머지입구를 모두 0으로 채운다. 끝의 두 입구는 0xc0000000에서 0xc07fffff사이의 선형주소령역에 해당한다. 0, 1, 0x300, 0x301번째 입구를 다음과 같이 초기화한다.

- 0과 0x300번째 입구의 주소마당을 pg0의 주소로, 1과 0x301번째입구의 주소마당을 pg1의 주소로 설정한다.
- Present, Read/Write, User/Supervisor기발을 모두 1로 설정한다.
- Accessed, Dirty, PCD, PWD, Page Size기발을 모두 0으로 설정한다.

startup_32()기호언어함수는 swapper_pg_dir의 물리주소를 cr3조종등록기에 저장하고 cr0조종등록기의 PG기발을 1로 설정해서 페지화유니트를 사용하게 만든다.

· **최종핵심부페지표**

핵심부는 최종적으로 0xc0000000부터 시작하는 선형주소를 0부터 시작하는 물리주소로 변환하도록 페지표를 만든다.

__pa마크로는 PAGE_OFFSET부터 시작하는 선형주소를 해당 물리주소로 변환하며 __va마크로는 그 반대일을 수행한다.

핵심부주페지대역등록부는 여전히 swapper_pg_dir에 들어있으며 paging_init()는 이것을 다음과 같이 초기화한다.

1. pagetable_init()을 호출해서 페지표입구를 정확히 설정한다.
2. swapper_pg_dir의 물리주소를 cr3조종등록기에 기록한다.
3. __flush_tlb_all()을 호출해서 모든 TLB입구를 비운다.

pagetable_init()함수가 하는 일은 체계에 있는 RAM의 크기와 CPU모형에 따라 다르다.

우선 가장 간단한 경우부터 시작하자. 콤퓨터에 있는 RAM크기가 896MB이하로 32bit물리주소로 모든 RAM의 주소를 지정할수 있어서 PAE수법(앞에서 본 《물리주소확장페지화수법》을 참고)을 사용할 필요가 없다고 하자.

다음과 같은 코드로 swapper_pg_dir 페지대역등록부를 다시 초기화한다.

```
    pgd_t *pgd_base = swapper_pg_dir;
 ...
    setup_identity_mappings(pgd_base, PAGE_OFFSET, end);
    remap_numa_kva();
    vaddr = __fix_to_virt(__end_of_fixed_addresses - 1) & PMD_MASK;
        fixrange_init(vaddr, 0, pgd_base);
```

end변수에는 3GB부터 시작하여 사용가능한 물리기억기의 끝에 해당하는 선형주소

가 들어있다. 여기서 CPU는 최신 80x86처리기이고 4MB크기의 페지와 《대역》 TLB 입구를 지원한다고 가정한다. 0xc0000000보다 큰 선형주소를 참조하는 모든 페지대역등록부입구의 User/Supervisor기발은 0이고 따라서 사용자프로쎄스는 핵심부주소공간에 접근할수 없다.

핵심부가 첫번째 단계의 초기화를 끝마치기 위해서 startup_32()함수에서 설정한 처음 8MB물리기억기를 같은 주소로 배치하는것이 필요하다. 이 배치가 더는 필요하지 않게 되면 핵심부는 zap_low_mappings()함수를 호출해서 해당 페지표입구를 지운다.

뒤에 나오는 《고정배치하는 선형주소》에서 보지만 핵심부는 고정배치하는 선형주소에 해당하는 페지표의 입구를 조정한다. 선형주소의 웃자리 128MB령역은 여러 종류의 배치을 위해 남겨둔다.(뒤에서 보게 되는 《고정배치하는 선형주소》와 《불련속적인 기억기령역관리》를 참고) 따라서 핵심부가 RAM을 배치하는데 사용할수 있는 공간은 1GB - 128MB = 896MB이다.

· RAM크기가 896MB에서 4096MB사이일 때 최종핵심부페지표

이 경우 RAM전체를 핵심부선형주소공간으로 배치할수 없다. Linux가 할수 있는 최선의 일은 초기화단계에서는 896MB만큼의 RAM구역을 핵심부선형주소공간으로 배치하고 프로그람의 다른 주소에 접근해야 할 때에는 다른 선형주소구역을 요청한 RAM으로 배치해야 한다. 이것은 페지표입구를 바꾸는것을 필요로 한다. 이렇게 동적으로 다시 배치하는 방법은 다음에 론의한다.

페지대역등록부를 초기화할 때 핵심부는 앞에 나온것과 같은 코드를 사용한다.

· RAM크기가 4096MB이상일 때 최종핵심부페지표

이제 4GB이상의 RAM을 장착한 콤퓨터에서 핵심부페지표를 초기화하는것을 보자. 더 정확히 말하면 다음과 같은 경우에만 이와 같이 처리한다.

- CPU가 물리주소확장을 지원하는 모형일 때
- RAM의 크기가 4GB이상일 때
- 핵심부를 PAE를 지원하도록 콤파일했을 때

PAE로 36bit물리주소를 다룰수 있지만 선형주소는 여전히 32bit주소이다. 앞의 경우와 마찬가지로 896MB의 RAM구역을 핵심부선형주소공간에 배치한다. 나머지 RAM은 배치하지 않은 상태로 남겨두고 동적재배치를 통해서 사용한다. 앞의 경우와 다른점은 3단계 페지화모형을 사용한다는 사실이다. 따라서 페지대역등록부를 아래와 같이 초기화한다.

```
  pgd_t *pgd_base = swapper_pg_dir;
for (i = 0; i < PTRS_PER_PGD; i++) {
pmd_t *pmd = (pmd_t *) alloc_bootmem_low_pages(PAGE_SIZE);
set_pgd(pgd_base + i, __pgd(__pa(pmd) + 0x1));
}
```

```
    ...
    setup_identity_mappings(pgd_base, PAGE_OFFSET, end);
  remap_numa_kva();
  vaddr = __fix_to_virt(__end_of_fixed_addresses - 1) & PMD_MASK;
  fixrange_init(vaddr, 0, pgd_base);
```

핵심부는 사용자선형주소공간에 해당하는 페지대역등록부의 처음 세 입구를 빈페지(empty_zero_page)의 주소로 초기화하고 4번째 입구는 페지중간등록부(pmd)의 주소로 초기화한다. 페지중간등록부의 처음 448개 입구(실제로 입구가 512개 있지만 끝에서 64개의 입구는 불련속적인 기억기할당을 위해 예약되여있다.)를 RAM에서 처음 896MB의 물리주소로 채운다.

모든 CPU모형에서 PAE나 2MB크기의 페지대역페지를 지원하는것은 아니다. 앞의 경우와 마찬가지로 Linux는 가능하면 큰 페지를 사용하여 페지표의 개수를 줄이려고 한다.

4) 고정배치하는 선형주소

핵심부선형주소의 마지막 1GB의 앞부분은 체계에 있는 물리기억기로 배치되는것을 보았다. 그렇지만 적어도 128MB의 선형주소는 핵심부가 불련속적인 기억기할당과 고정배치하는 선형주소를 구성하는데 사용할수 있도록 항상 비워둔다.

불련속적인 기억기할당이란 단지 동적으로 기억기페지를 할당하고 해제하는 특별한 방식으로서 《불련속적인 기억기령역관리》에서 설명한다. 여기서는 고정배치하는 선형주소에만 초점을 두고 설명한다.

기본적으로 《고정배치하는 선형주소(fix_mapped linear address)》란 0xffffdf0와 같은 상수값을 포함하는 선형주소로서 이에 해당하는 물리주소는 임의의 방법으로 설정할수 있다. 따라서 매 고정배치하는 선형주소는 물리기억기의 한 페지틀으로 배치된다.

고정배치하는 선형주소는 개념적으로 보면 처음 896MB의 RAM으로 배치하는 선형주소와 비슷하다. 그렇지만 고정배치하는 선형주소는 어떤 물리주소로도 배치할수 있는데 대해 마지막 1GB의 앞부분에 있는 배치는 련속적이다.(선형주소 X를 물리주소 X_PAGE_OFFSET으로 배치한다.)

지적자변수를 사용하는것보다 고정배치하는 선형주소가 조금 더 효률적이다. 사실 지적자변수가 가리키는곳을 참조할 때에는 상수로 된 주소를 참조할 때 기억기를 한번 더 접근해야 한다. 나아가서 지적자변수를 참조하기 전에 값이 정확한가를 확인하는것이 좋은 프로그람작성습관이지만 상수인 선형주소를 사용하는 경우에는 이런 검사가 필요없다.

모든 고정배치하는 선형주소는 enum fixed_addresses자료구조로 정의하고 옹근수 색인으로 표현한다.

```
enum fixed_addresses {
        FIX_HOLE,
    FIX_VSYSCALL,
     #ifdef CONFIG_X86_LOCAL_APIC
     FIX_APIC_BASE,
#endif
    #ifdef CONFIG_X86_IO_APIC
        FIX_APIC_BASE_0,
     [. . .]
     __end_of_fixed_addresses
};
```

고정배치하는 선형주소는 선형주소의 마지막 1GB령역의 끝에 위치한다. fix_to_virt( )함수는 색인을 가지고 상수선형주소를 계산한다.

```
static __always_inline unsigned long
fix_to_virt(const unsigned int idx)
{
 if (idx >=__end_of_ fixed_addresses)
          __this_fixmap_does_not_exist( );
 return __fix_to_virt(idx);
}
```

어떤 핵심부함수가 fix_to_virt(FIX_IOAPIC_BASE_0)를 호출하였다고 하자. 이 함수를 직결(inlne)으로 정의하고있기때문에 C콤파일러는 fix_to_virt()함수를 호출하지 않고 이것을 리용하는 함수에 코드를 단순히 삽입한다. 실행할 때에는 색인값을 검사하지 않는다.

FIX_IOAPIC_BASE_0은 상수이고 콤파일러는 콤파일시 if문장의 조건이 거짓이라는 사실을 알기때문에 해당 문장을 잘라낼수 있다. 반대로 if문장의 조건이 참이거나 fix_to_virt()함수의 파라메터가 상수가 아니라면 __this_fixmap_does_not_exist라는 기호를 어디서도 정의하지 않으므로 련결하는 단계에서 오유가 발생한다. 결론적으로 콤파일러는 (FIXADDR_TOP - ((x) << PAGE_SHIFT))값을 계산해서 fix_to_virt() 함수호출을 상수선형주소인 0xffffe000으로 대체한다.

핵심부는 set_fixmap(idx,phys)과 set_fixmap nocache(idx,phys)함수를 사용해서 물리주소를 고정배치하는 선형주소와 련결한다. 두 함수는 모두 fix_to_virt(idx) 선형주소에 해당하는 페지표입구를 초기화한다. 그렇지만 뒤의 함수는 페지표입구의 PCD기발을 1로 설정하여 해당 페지틀에 있는 자료를 접근할 때 하드웨어캐쉬를 끄게 한다.(앞서 설명한 《하드웨어캐쉬》를 참고)

6. 하드웨어캐쉬와 TLB다루기

최근 콤퓨터구성방식에서 하드웨어캐쉬와 변환참조완충기(TLB)는 성능을 높이는데서 중요한 역활을 한다. 그래서 핵심부개발자들은 캐쉬실패나 TLB실패의 회수를 줄이려고 여러가지 기법을 사용한다.

1) 하드웨어캐쉬다루기

처음에 이야기한바와 같이 하드웨어캐쉬는 캐쉬선로단위로 접근한다. L1_CACHE_BYTE마크로는 캐쉬선로의 바이트단위크기이다. 이 값은 펜티움4 이전 모형에서는 32, 펜티움4에서는 128이다.

캐쉬명중비률을 높이기 위하여 핵심부는 구성방식를 고려하여 다음과 같은 결정을 내린다.

- 자료구조에서 가장 자주 사용하는 마당을 자료구조의 앞부분에 두어 캐쉬에서 같은 선로에 들어가도록 한다.
- 핵심부는 큰 크기의 자료구조를 할당할 때 캐쉬선로를 균등하게 사용할수 있는 형태로 기억기에 저장한다.
- 핵심부는 프로쎄스절환을 할 때 이전에 동작하던 프로쎄스와 같은 페지표를 사용하는 프로쎄스를 더 먼저 호출한다.(《schedule( )함수》를 참고)

2) TLB다루기

일반적으로 어떤 프로쎄스절환이 일어나든 현재 사용하는 페지표를 바꾸어야 한다. 이전의 페지표를 가지고 만든 국부 TLB 역시 모두 비워야 한다. 이 일은 핵심부가 cr3 조종등록기에 새로운 페지대역등록부의 주소를 쓸 때 자동으로 일어난다. 가끔 다음과 같은 경우 핵심부가 TLB를 비우지 않아도 된다.

- 똑같은 페지표를 사용하는 2개의 일반프로쎄스사이에서 프로쎄스절환을 하는 경우 (《schedule()함수》를 참고)
- 일반프로쎄스와 핵심부스레드사이에서 프로쎄스절환을 하는 경우

《핵심부스레드의 기억기서술자》에서 보았지만 핵심부스레드는 자기만의 페지표가 없으며 핵심부스레드를 실행하는 CPU에서 마지막으로 실행된 일반프로쎄스의 페지표를 사용한다.

프로쎄스절환외에도 핵심부가 TLB에 있는 일부 입구를 비워야 하는 경우가 있다. 례를 들어 핵심부가 사용자방식프로쎄스에 페지틀을 할당한 후 해당 물리주소를 페지표입구에 기록하면 해당 선형주소를 참조하는 모든 국부TLB입구를 비워야 한다. 다중처리기체계에서 핵심부는 같은 페지표를 사용하는 CPU가 있는 경우 해당 CPU에 있는 같은 TLB입구도 비워야 한다.

핵심부는 다음 함수와 마크로를 사용하여 TLB입구를 비운다.

__flush_tlb_one

지정한 주소를 포함하는 페지에 대한 국부 TLB입구를 비운다.

flush_tlb_page

모든 CPU에서 지정한 주소를 포함하는 페지에 대한 국부 TLB입구를 비운다. 이를 위해 핵심부는 다른 CPU에 처리기사이 새치기(interprocessor interrupt)를 보낸다.(《처리기간새치기처리》를 참고)

local_flush_tlb와 __flush_tlb

현재프로쎄스의 모든 페지에 대한 국부 TLB를 비운다. 이를 위해 cr3등록기의 현재값을 읽은 후 다시 같은 값을 쓴다. 펜티움 pro이후의 프로쎄스에서는 대역이 아닌 페지(Global 기발이 0인 페지)의 TLB입구만 비운다.

flush_tlb

모든 현재프로쎄스의 대역이 아닌 페지에 대한 TLB입구를 비운다. 이 과정에서 모든 CPU는 __flush_tlb를 호출하도록 하는 처리기간 새치기를 받는다.

flush_tlb_mm

지정한 페지표의 대역이 아닌 모든 페지에 대한 TLB입구를 비운다.(《기억기서술자》를 참고) 다음에 보지만 다중처리기체계에서 CPU두개 이상이 똑같은 페지표를 공유하는 프로쎄스를 실행할수도 있다. 80x86구성방식에서 이 함수는 모든 CPU에서 지정한 페지표의 대역이 아닌 모든 페지에 대한 국부 TLB입구를 비우도록 한다.

제 2 절. 기억기관리

1절에서는 Linux에서 80x86의 토막화와 페지화회로를 리용하여 론리적인 주소를 물리적인 주소로 변환하는것을 보았다. 또한 주기억의 일부분을 영구적으로 핵심부에 할당하여 핵심부코드와 핵심부의 정적자료구조를 저장하는데 사용한다는 점도 언급하였다.

주기억의 남은 부분을 동적기억기(dynamic memory)라고 부른다. 사실 전체 체계의 성능은 동적기억기를 얼마나 효률적으로 관리하는가에 달려있다. 따라서 현재의 모든 다중과제조작체계는 동적기억기사용을 최적화하고 필요할 때에만 할당하며 가능한 빨리 해제하려고 한다.

1. 페지틀관리

1절에서의 하드웨어페지화에서 Intel펜티움처리기가 페지틀의 크기로 4kB와 4MB 또한 PAE를 사용하는 경우 2MB라는 서로 다른 두 크기를 사용하는 방법을 보았다. Linux는 이것들중 작은 페지틀크기인 4kB를 표준 기억기할당단위로 채택한다. 이것은 다음 두가지 리유로 작업을 간단하게 만들어준다.

· 페지화회로가 발생시킨 페지절환례외를 쉽게 해석할수 있다. 페지가 존재하지만 프로쎄스가 해당 페지에 접근할 권한이 없거나 페지가 존재하지 않는 경우 페지절환이 발생한다. 후자의 경우 기억기할당자는 사용가능한 4kB페지틀을 찾아서 프로쎄스에 할당해야 한다.

· 4kB는 대부분의 디스크블로크크기의 배수이다. 따라서 주기억기와 디스크사이의 자료전송이 더 효률적이다. 아직까지는 4kB크기가 4MB크기보다는 다루기 쉽다.

1) 페지서술자

핵심부는 매 페지틀의 현재상태를 계속 유지해야 한다. 례를 들어 프로쎄스에 소속된 페지를 포함한 페지틀과 핵심부코드나 핵심부자료구조체를 포함한 페지틀을 구별할수 있어야 한다. 마찬가지로 동적기억기에 있는 페지틀이 사용중인지 아닌지 알수 있어야 한다. 동적기억기에 있는 페지틀에 쓸모있는 자료가 들어있지 않다면 이 페지틀은 사용중이 아니다. 페지틀에 사용자방식프로쎄스의 자료나 쏘프트웨어 캐쉬자료, 동적으로 할당한 핵심부자료구조체, 장치구동프로그람의 완충용자료, 핵심부모듈의 코드 등이 들어있다면 이 페지틀은 사용중이다.

struct page형인 페지서술자에 페지틀의 상태정보를 보관한다. 여기에는 표 4-2에 나오는 마당이 들어있다. 모든 페지서술자를 mem_map배렬에 보관한다. 매 서술자의 크기는 64byte보다 작기때문에 mem_map을 위해 주기억 1MB당 페지틀 4개가 필요하다.

표 4-2. 페지서술자에 있는 마당

형	이 름	설 명
struct address_space *	mapping	페지를 페지캐쉬에 삽입할 때 사용한다.(《페지캐쉬》 참고)
pqoff_t	index	페지의 디스크영상안에서 페지가 저장하고있는 자료의 위치이거나 교환하여 내보내기(swap out)페지식별자이다.
atomic_t	_mapcount	페지표의 참조회수이다.
atomic_t	_count	페지의 참조회수이다.
page_flags_t	flags	기발의 배렬이다.(표 4-3 참고)
struct list_head	lru	오래동안 사용하지 않은 페지의 2중련결목록에 대한 지적자를 담는다.
unsigned long	private	이 페지가 완충기를 저장하고있을 때 사용한다.(《페지입출력연산》를 참고)
void *	Virtual	페지틀의 마지막 1GB령역에서의 선형주소이다.(《페지틀 요청과 해제》를 참고)

우의 마당들중 일부는 페지틀이 사용중인가, 어떤 핵심부구성요소가 해당 페지틀을 사용하는가에 따라 의미가 다르기때문에 여기서는 두 마당만 자세히 설명한다.

_count

해당 페지의 사용참조회수이다. 이 값이 0이면 해당 페지틀을 사용하지 않고있으며 어떤 프로쎄스에나 또는 핵심부 자기에 할당할수 있다. 한편 값이 0보다 크면 페지틀을 프로쎄스 하나이상에 할당하거나 어떤 핵심부자료구조체를 보관하기 위해 사용하고있다.

flags

페지틀의 상태를 서술하는 기발을 최대 32개까지 포함하는 배렬이다.(표 4-3 참고)

핵심부는 매 PG_xyz기발마다 해당 값을 다루는 마크로를 몇가지 정의한다. 일반적으로 PageXyz마크로는 기발의 값을 반환하고 SetPageXyz와 ClearPageXyz마크로는 각각 해당 비트를 1로 설정하거나 0으로 지운다.

표 4-3. 페지틀의 상태를 서술하는 기발

기 발 명	의 미
PG_locked	디스크입출력연산과 관련된 페지이다.
PG_error	페지전송중 입출력오유가 발생하였다.
PG_referenced	디스크입출력연산을 위해 이 페지를 최근에 접근하였다.
PG_uptodate	디스크입출력오유가 발생하지 않고 읽기작업을 마친 후에 이 기발을 설정한다.
PG_dirty	페지를 수정하였다.(《try_to_swap_out()함수》를 참고)
PG_lru	페지가 활성 또는 비활성페지목록에 들어있다.(《오래동안 사용하지 않은 목록》참고)
PG_active	페지가 활성페지목록에 들어있다.(《오래동안 사용하지 않은 목록》참고)
PG_slab	페지틀이 스래에 들어있다.(《기억기령역관리》참고)
PG_skip	사용하지 않는다.
PG_highmem	페지틀이 ZONE_HIGHMEM령역에 속한다.(《기억기령역》참고)
PG_checked	Ext2파일체계에서 사용하는 기발이다.
PG_arch_1	80x86기본방식에서는 사용하지 않는다.
PG_reserved	핵심부코드용으로 예약했거나 사용할수 없는 페지틀이다.
PG_launder	shrink_cache()에 의해 일어난 입출력연산과 관련된 페지이다.(《shrink_cache()함수》를 참고)

2) 기억기령역

리상적인 콤퓨터구조에서 페지들은 어떤 용도로도(핵심부와 사용자자료를 보관하고 디스크자료를 완충시키는 등) 사용가능한 기억기저장단위이다. 어떤 종류의 자료를 담는 페지이든 아무런 제한없이 어떤 페지틀에나 저장할수 있다. 하지만 실제 콤퓨터구조에는 페지틀을 사용하는 방식을 제한하는 하드웨어 제약이 있다. 특히 Linux는 80x86기본 방식에 있는 두가지 하드웨어제약을 처리해야 한다

· ISA모선용직접기억기접근(DMA:Direct Memory Access)처리기는 주기억의 처음 16MB만 접근할수 있다는 강한 제한이 있다.

· 많은 주기억을 가지는 최근의 32bit콤퓨터에서는 선형주소공간이 너무 작아서 CPU가 모든 물리기억기에 직접 접근할수 없다.

Linux는 이런 제한을 다루려고 물리기억기를 세개 령역(zone)으로 쪼갠다.

ZONE_DMA

16MB아래의 기억기페지를 포함한다.

ZONE_NORMAL

16MB부터 896MB까지의 기억기페지를 포함한다.

ZONE_HIGHMEM

896MB이상의 기억기페지를 포함한다.

ZONE_DMA령역은 오래된 ISA기반장치에서 DMA를 리용할 때 사용할수 있는 기억기페지를 포함한다.(직접기억기접근에서 DMA를 자세히 설명한다).

ZONE_DMA와 ZONE_NORMAL령역은 핵심부가 선형주소공간의 마지막 1GB로 선형배치하여 직접 접근할수 있는 일반 기억기페지를 포함한다. 반면에 ZONE_HIGHMEM령역은 핵심부가 선형주소공간의 마지막 1GB로 선형배치하여 직접 접근할수 없는 기억기페지를 포함한다. 64bit구조에서는 ZONE_HIGHMEM령역을 사용하지 않는다.

각 기억기령역은 struct zone(zone_t라는 이름도 있다)형인 자기만의 서술자를 가진다. 표 4-4는 이 서술자에 있는 마당을 보여준다.

표 4-4. 령역서술자에 있는 마당

형	이 름	설 명
char *	name	령역공식이름((DMA와 Nomal, HighMem)을 가리키는 지적자를 담는다.
Spinlock_t	lock	서술자를 보호하는 잠그기
unsigned long	free_pages	이 령역에 있는 사용하지 않는 페지의 수
unsigned long	pages_min	이 령역에서 사용하지 않고 남겨두어야 하는 최소 페지의 수(《페지틀해제》참고)

unsigned long	pages_low	이 령역의 페지균형알고리듬에서 사용하는 림계값(《페지틀해제》참고)
unsigned long	pages_high	이 령역의 페지균형알고리듬에서 사용하는 높은 림계값(《페지틀해제》참고)
free_area_t []	free_area	형제체계페지할당자에서 사용한다.(《형제체계알고리듬》참고)
struct pglist_data *	zone_pgdat	이 령역에 속해있는 마디서술자를 가리키는 지적자
struct page *	zone_mem_map	이 령역에 있는 페지서술자의 배렬(《형제체계알고리듬》참고)
unsigned long	zone_start_pfn	이 령역의 시작물리주소

페지서술자의 zone마당은 해당 페지틀이 속한 령역의 서술자를 가리킨다.

zone_names 배렬은 세 령역의 공식이름(《DMA》와 《Nomal》, 《HighMem》)을 보관한다. 핵심부가 기억기 할당함수를 호출할 때 요청한 페지틀을 담을 령역을 지정해야 한다. 일반적으로 핵심부는 어떤 령역을 사용하고 싶은가를 지정한다. 례를 들어 선형주소공간의 마지막 1GB에 직접 배치되여있어야 하지만 ISA DMA전송용으로 사용하지 않을 페지틀이라면 핵심부는 ZONE_NORMAL이나 ZONE_DMA에 있는 페지틀을 요청한다. 응당 ZONE-NORMAL에 여유페지틀이 더는 없을 때에만 ZONE_DMA에서 페지틀을 할당해야 한다. 핵심부는 기억기할당을 요청할 때 요구하는 령역을 지정하는데 령역서술자지적자의 배렬인 struct zonelist(zonelist_t라는 이름도 있다.)자료구조를 사용한다.

3) 불균등기억기접근

일반적으로 콤퓨터의 기억기를 균등한(homogeneous)공유자원이라고 생각한다. 하드웨어캐쉬의 역할을 고려하지 않고 생각하면 한 CPU가 기억기위치에 접근하는데 걸리는 시간은 기본적으로 물리주소의 위치나 CPU에 상관없이 똑같다. 그러나 이런 가정이 맞지 않는 구조도 있다. 례를 들면 일부 다중처리기 Alpha나 MIPS체계가 그렇다. Linux에서는 불균등기억기접근(NUMA:Non_Uniform Memory Access)방식을 지원한다. 여기서는 한 CPU에서 다른 기억기위치에 접근하는데 걸리는 시간이 틀릴수 있다.

체계에 있는 물리기억기는 여러 마디(node)로 쪼개져 들어간다. 어떤 CPU가 한 마디안에 있는 페지에 접근하는데 걸리는 시간은 똑같지만 이 시간은 서로 다른 두 CPU 사이에서는 틀릴수 있다. 핵심부는 어떤 CPU가 가장 자주 접근하는 핵심부자료구조체를 보관하는 위치를 조심스럽게 선택하여 모든 CPU가 시간이 오래 걸리는 마디에 접근하는 수를 최소화하려고 한다. 앞에서 본바와 같이 매 마디에 있는 물리기억기를

여러 령역으로 쪼갤수 있다. 매 마디는 pg_data_t형서술자를 포함한다. 표 4-5는 이 서술자에 있는 마당을 보여준다. 모든 마디서술자를 단순히 련결목록에 보관하며 pgdat_list변수는 이 목록의 첫번째 항목을 가리킨다.

표 4-5. 마디서술자에 있는 마당

형	이 름	설 명
zone_t[]	node_zones	마디에 있는 령역서술자의 배렬
zonelist_t[]	node_zonelists	페지할당자가 사용하는 zonelist_t자료구조의 배렬 (《페지를 요청과 해제》참고)
int	nr_zones	이 마디에 있는 령역의 수
struct page *	node_mem_map	마디에 있는 페지서술자의 배렬
struct bbotmem_data *	bdata	핵심부초기화 단계에서 사용
unsigned long	node_start_pfn	마디의 시작물리주소
unsigned long	node_size	마디의 크기(페지의 수)
int	node_id	마디의 식별자
pg_data_t *	node_next	마디목록의 다음항목

마찬가지로 여기서는 주로 80x86기본방식에 대하여 본다. IBM호환PC에서는 균등기억기접근(UMA:Uniform Mcmory Access)방식을 사용하기때문에 NUMA지원은 실제로 필요없다. 그렇지만 핵심부콤파일을 할 때 NUMA지원을 빼더라도 Linux는 체계에 있는 모든 물리기억기를 포함하는 마디 하나를 사용한다. 해당 서술자는 contig_page_data변수에 들어있다.

80x86기본방식에서 물리기억기(physical memory)를 마디 하나로 모으는것은 쓸데없는 일처럼 보일수 있다. 그렇지만 이런 접근은 핵심부가 모든 구조에서 물리기억기가 마디 하나 이상에 나누어 들어간다고 가정할수 있으므로 기억기관리를 하는 코드의 호환성을 높여준다.

4) 기억기관리자료구조체의 초기화

그림 4-13은 동적기억기와 이것을 참조하는데 사용하는 값을 보여준다. 기억기의 여러 령역을 일정한 비례로 그렸다.

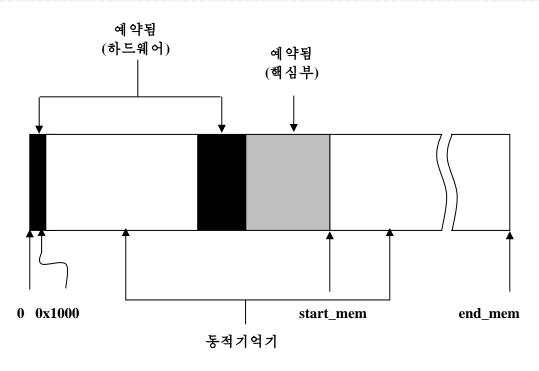



그림 4-13. 기억기배치

이미 1절에 있는 《핵심부페지표》에서 paging_init()함수가 체계에 있는 주기억의 크기에 따라 어떻게 핵심부페지표를 초기화하는가에 대하여 설명하였다. paging_init()함수는 페지표외에 기억기를 다루는 다른 자료구조체도 초기화한다. 이 함수는 kmap_init()함수를 호출하여 핵심부가 ZONE_HIGHMEM령역에 접근할수 있도록 선형주소의 창(window)을 만드는 kmap_pte변수를 설정한다.(《림시핵심부배치》를 참고) 그후 3가지 기억기령역의 크기를 보관하는 배렬을 파라메터로 해서 free_area_init() 함수를 호출한다.

free_area_init()함수는 령역서술자와 페지틀서술자를 모두 초기화한다. 이 함수는 각 기억기령역의 크기를 나타내는 zones_size배렬을 파라메터로 받고 다음과 같은 작업을 수행한다.

1. zones_size에 있는 값을 모두 더해서 주기억에 있는 전체 페지틀의 수를 계산하고 그 결과를 국부변수 totalpages에 보관한다.

2. 페지서술자의 active_list와 inactive_list목록을 초기화한다.

3. 페지서술자의 mem_map배렬용으로 공간을 할당한다. totalpages에 페지서술자의 크기를 곱한 크기만큼의 공간이 필요하다.

4. contig_page_data마디서술자에 있는 마당 몇개를 초기화한다.

calculate_zone_totalpages(pgdat, zones_size, zholes_size);

5. 모든 페지서술자의 마당 몇개를 초기화한다.

node_alloc_mem_map(pgdat);

6. 기억기령역서술자의 주소를 국부변수 zone에 보관하고 zone_names배렬에 있는 매 항목마다(j는 0에서 2까지) 다음 단계를 수행한다.

a. 서술자의 마당 몇개를 초기화한다.

b. 령역이 비여있다면(즉 페지틀이 하나도 없으면) 6단계의 처음으로 되돌아가 다음령역으로 넘어간다.

c. 령역에 페지틀이 하나라도 있으면 령역서술자의 pages_min과 pages_low, pages_high마당을 초기화한다.

d. 령역서술자의 zone_mem_map마당을 령역에 있는 첫번째 페지서술자의 주소로 설정한다.

e. 령역서술자의 zone_start_pfn 마당을 령역에 있는 첫번째 페지틀의 물리주소로 설정한다.

f. 령역내에 있는 모든 페지틀의 페지서술자에 있는 zone마당에 령역서술자의 주소를 저장한다.

g. 령역이 ZONE_DMA이나 ZONE_NORMAL이라면 령역내에 있는 모든 페지서술자의 virtual마당에 페지틀을 배치하는 마지막 1GB공간에서의 선형주소를 저장한다.

h. 령역서술자의 free_area배렬에 있는 free_area_t구조체를 초기화한다(《형제체계알고리듬》을 참고).

paing_init()함수를 마칠 때에는 모든 페지에 PG_reserved기발이 설정된 상태이기때문에 아직 동적기억기를 사용할수 없다. paging_init()을 실행한 후에 호출하는 mem_init( )함수에서 추가로 기억기초기화작업을 수행한다.

기본적으로 mem_init()함수는 체계에 존재하는 전체 페지틀수인 num_physpages의 값을 초기화한다. 다음으로 동적기억기에 속하는 모든 페지틀을 훑어본다. 매 페지틀마다 해당 서술자의 count마당을 1로 설정하고 PG_reserved기발을 지우고 페지틀이 ZONE_HIGHMEM령역에 속하면 PG_highmem기발을 설정한 후 이 페지틀에 대해 nr_free_page()함수를 호출한다. nr_free_page()는 페지틀을 해제하는것외에도(뒤에 나오는 《 형제체계알고리듬 》을 참고) 해당 페지틀을 소유한 기억기령역서술자의 free_pages마당의 값도 증가시킨다. 모든 령역서술자에 있는 free_pages마당은 nr_free_pages()함수에서 동적기억기에 있는 사용하지 않는 전체 페지틀수를 계산하는데 사용한다.

mem_init()함수는 동적기억기에 속하지 않는 페지틀의 수도 센다. 핵심부를 콤파일할 때 만들어지는 여러 기호에 의해 하드웨어와 핵심부코드, 핵심부자료용으로 예약된 페지틀의 수와 핵심부초기화과정에서 사용하고 초기화를 마친후 해제할수 있는 페지틀의 수를 계산할수 있다.(《예약된 페지틀》을 참고)

5) 페지틀요청과 해제

조금씩 다른 함수 6개와 마크로를 리용하여 페지틀을 요청할수 있다. 별도로 언급하지 않으면 할당한 첫번째 페지의 선형주소나 할당이 실패한 경우 NULL을 반환한다.

alIoc_pages(gfp_mask, order)

련속된 페지틀 2^{order}개를 요청할 때 사용하는 함수이다. 할당한 첫번째 페지의 선형주소나 할당이 실패한 경우 NULL을 반환한다.

alloc_page(gfp_mask)

페지틀 하나를 얻을 때 사용하는 마크로이다. 이 마크로는 다음과 같이 변환된다.

alloc_page(gfp_mask,0)

이 마크로는 할당한 페지틀의 서술자주소나 할당이 실패한 경우 NULL을 반환한다.

__get_free_pages(gfp_mask, order)

alloc_pages()와 비슷하지만 할당한 첫번째 페지틀의 선형주소를 반환한다.

__get_free_page(gfp_mask)

페지틀 하나를 얻을 때 사용하는 마크로이다. 이 마크로는 다음과 같이 변환된다.

__get_free_pages(gfp_mask, 0)

get_zeroed_page(gfp_mask) 또는 이와 동일한 get_free_page(gfp_mask)

이 함수는 다음을 호출한다.

alloc_pages(gfp_mask, 0)

다음으로 할당받은 페지틀을 0으로 채운다.

__get_dma_pages(gfp_mask, order)

DMA용으로 적합한 페지틀을 얻을 때 사용하는 마크로이다. 이 마크로는 다음과 같이 변환된다.

__get_free_pages(gfp_mask 1__GFP_DMA, order)

gfp_mask는 여유페지틀을 어떻게 찾겠는가를 지정하며 다음기발로 구성된다.

__GFP_ WAlT

핵심부는 사용하지 않는 페지틀이 생기기를 기다리며 현재프로쎄스를 차단할수 있다.

__GFP_ HlGH

핵심부는 기억기가 아주 적게 남은 상태에서 복구하려고 남겨둔 여유페지틀의 보관고에 접근할수 있다.

__GFP_ IO

핵심부는 페지틀을 해제하기 위해 낮은 주소에 있는 기억기페지에 대해 입출력전송을 수행할수 있다.

__GFP_HlGHlO

핵심부는 페지틀을 해제하기 위해 높은 주소에 있는 기억기페지에 대해 입출력전송을 수행할수 있다.

__GFP_ FS

핵심부는 저수준 VFS연산을 수행할수 있다.

__GFP_DMA

ZONE_DMA령역에 들어있는 페지들을 할당해야 한다(앞에서 본 《기억기령역》을 참고).

__GFP_HIGHMEM

ZONE_HIGHMEM령역에 들어있는 페지들을 할당해도 된다.

Linux는 실제로 이 기발을 개별적으로 사용하기 보다는 표 4-6에 렬거한 미리 정의한 기발조합을 사용한다. 페지할당함수 6개를 호출하면서 파라메터를 전달할 때 실제로 마주치는것은 이 그룹명이다.

표 4-6. 페지들을 요청할 때 사용하는 기발값의 그룹

그 룹 명	해 당 기 발
GFP_ATOMIC	__GFP_HIGH
GFP_NOIO	__GFP_HIGH __GFP_WAIT
GFP_NOHIGHIO	__GFP_HIGH __GFP_WAIT __GFP_IO
GFP_NOFS	__GFP_HIGH __GFP_WAIT __GFP_IO __GFP_HIGHIO
GFP_KERNEL	__GFP_HIGH __GFP_WAIT __GFP_IO __GFP_HIGHIO __GFP_FS
GFP_NFS	__GFP_HIGH __GFP_WAIT __GFP_IO __GFP_HIGHIO __GFP_FS
GFP_KSWAPD	__GFP_WAIT __GFP_IO __GFP_HIGHIO __GFP_FS
GFP_USER	__GFP_WAIT __GFP_IO __GFP_HIGHIO __GFP_FS
GFP_HIGHUSER	__GFP_WAIT __GFP_IO __GFP_HIGHIO __GFP_FS __GFP_HIGHMEM

__GFP_DMA와 __GFP_HIGHMEM기발을 가리켜 령역변경자(zone modifier)라고 하는데 사용하지 않는 페지들을 찾을 때 핵심부가 검색할 령역을 지정한다. contig_page_data마디서술자의 node_zonelists마당은 령역서술자목록의 배렬이다. 각 목록은 령역변경자의 특정조합 하나와 련결된다. 이 배렬에는 항목이 16개 있지만 령역변경자가 2개뿐이므로 실제로 4개만 사용한다. 이것을 표 4-7에서 보여준다.

표 4-7. 령역변경자목록

GFP DMA	GFP HIGHMEM	령 역 목 록
0	0	ZONE_NORMAL + ZONE_DMA
0	1	ZONE_HIGHMEM + ZONE_NORMAL + ZONE_DMA
1	0	ZONE_DMA
1	1	ZONE_DMA

페지틀을 해제할 때에는 다음 4개 함수나 마크로중 하나를 사용할수 있다.

__free_pages(page, order)

이 함수는 page가 가리키는 페지서술자를 검사한다. 그래서 예약된 페지틀이 아니면(즉 PG_reserved기발이 0이면) 서술자의 count마당을 감소시킨다. count가 0이면 page부터 시작하는 련속된 페지틀 2^{order}개를 사용중이 아니라고 간주한다. 이 경우 이 함수는 __free_pages_ok()를 호출하여 첫번째 여유페지의 페지틀서술자를 적절한 여유페지틀의 목록에 추가한다.

free_pages(addr, order)

이 함수는 __free_pages()와 비슷하지만 해제할 첫 페지틀의 선형주소 addr을 파라메터로 받는다는 차이가 있다.

__free_page(p)

이 마크로는 page가 가리키는 서술자를 포함하는 페지틀을 해제하며 다음코드로 확장된다.

__free_pages(page, 0)

free_page(addr)

이 마크로는 선형주소 addr을 포함하는 페지틀을 해제하며 다음코드로 확장된다.

free_pages(addr, 0)

6) 웃자리기억기페지틀의 핵심부배치

896MB경계우에 있는 페지틀은 핵심부선형주소공간의 마지막 1GB로 배치되지 않으므로 핵심부는 이 페지틀에 직접 접근할수 없다. 즉 할당한 페지틀의 선형주소를 반환하는 모든 페지할당자함수는 웃자리기억기(high memory)에서 동작하지 않는다.

례를 들어 핵심부가 __get_free_pages(GFP_HIGHMEM, 0)를 호출하여 웃자리기억기에 있는 페지틀을 할당한다고 하자. 할당자가 웃자리기억기에 있는 페지틀을 할당하면 __get_free_pages()는 해당 페지틀의 선형주소가 존재하지 않아서 반환할수 없으므로 NULL을 반환한다. 게다가 핵심부는 해당 페지를 추적할수 없으므로 할당한 페지틀을 해제할수 없다.

간단히 말해서 웃자리기억기페지틀을 할당할 때에는 alloc_pages()함수와 더 간단

한 alloc_page()함수만을 사용해야 한다. 두 함수는 할당한 첫번째 페지틀의 페지서술자의 주소를 반환한다. 일단 할당한 후에는 해당 페지틀의 물리주소가 4GB를 넘는다고 하더라도 선형주소공간의 마지막 1GB로 배치해야 한다.

이를 위해 핵심부는 영구핵심부배치(permanent kernel mapping), 림시핵심부배치(temporary kernel mapping), 불련속적인 기억기할당(noncontiguous memory allocation)이라는 3가지 서로 다른 방법을 사용한다. 여기서는 처음 두 기법에 초점을 두고 세번째 기법은 후에 《불련속적인 기억기령역관리》에서 고찰한다.

영구핵심부배치을 할 때에는 현재프로쎄스를 차단할수도 있다. 이것은 웃자리기억기에 있는 페지틀에 대한 창으로 사용할 여유페지표입구점이 없을 때 일어날수 있다. 따라서 새치기처리기나 미룰수 있는 함수는 영구핵심부배치를 사용할수 없다. 반면에 림시핵심부배치는 절대로 현재프로쎄스를 차단하지 않는다. 그러나 부족점은 매우 적은 수의 림시핵심부배치만 동시에 할수 있다는점이다.

물론 이 방법중 어느것도 주기억전체에 동시에 접근할수 있게 하지 않는다. 결국 PAE는 체계가 주기억을 최대 64GB까지 가질수 있게 지원하지만 웃자리기억기를 배치할수 있는 선형주소공간은 128MB뿐이다.

◦ **영구핵심부배치**

영구핵심부배치는 핵심부가 웃자리기억기 페지틀을 핵심부주소공간으로 오래동안 배치할수 있게 한다. 영구핵심부배치는 전용 페지표을 사용한다. 이 페지표의 주소는 pkmap_page_table변수에 들어있다. LAST_PKMAP마크로는 이 페지표에 들어가는 입구점의 수를 지정한다. 알고있는것처럼 페지표에는 PAE사용여부에 따라 입구점이 512개나 1024개 들어간다.(1절의 《물리주소확장페지화기구》를 참고) 따라서 핵심부는 한번에 2MB나 4MB의 웃자리기억기에 접근할수 있다. 이 페지표는 PKMAP_BASE(일반적으로 0xff600000)부터 시작하는 선형주소를 배치한다. 웃자리기억기에 있는 첫번째 페지틀에 해당하는 서술자의 주소는 highmem_start_page변수에 들어있다.

pkmap_count배렬은 pkmap_page_table페지표에 있는 매 입구점마다 하나씩 LAST_PKMAP개의 계수기를 포함한다. 이 계수기의 값에 따라 3가지 경우로 나눌수 있다.

계수기0

해당 페지표 입구점는 어떤 웃자리 기억기페지틀도 배치하지 않고 사용할수 있다.

계수기 1

해당 페지표입구점은 어떤 웃자리 기억기페지틀도 배치하지 않지만 이것을 마지막으로 사용한 후에 해당 TLB입구점을 비우지 않아서 사용할수 없다.

계수기 n(n > 1)

해당 페지표입구점은 웃자리 기억기페지틀을 배치하며 핵심부구성요소 n-1개에서 사용한다. kmap() 함수는 영구핵심부배치을 만든다. 이것은 기본적으로 다음코드와 동

등하다.

```
void *kmap(struct page *page)
{
        might_sleep();
        if (page < highmem_start_page)
            return page_address(page);
        return kmap_high(page);
}
```

페지서술자의 virtual마당은 페지틀을 배치하는 마지막 1GB내의 선형주소가 있으면 이것을 저장한다. 따라서 896MB경계밑에 있는 모든 페지틀에서 virtual마당은 페지틀의 물리주소에 PAGE_OFFSET을 더한 값을 담는다. 반면에 웃자리기억기에 있는 페지틀에서 이 마당은 페지틀을 영구핵심부배치나 림시핵심부배치로 하는 경우에만 NULL이 아닌 값을 담는다.

kmap()함수는 페지틀이 실제로 웃자리기억기에 속한 경우 kmap_high()함수를 호출한다. 이 함수는 기본적으로 다음코드와 동등하다.

```
void fastcall *kmap_high(struct page *page)
{
   unsigned long vaddr;

   spin_lock(&kmap_lock);
   vaddr = (unsigned long)page_address(page);
   if (!vaddr)
        vaddr = map_new_virtual(page);
   pkmap_count[PKMAP_NR(vaddr)]++;
   if (pkmap_count[PKMAP_NR(vaddr)] < 2)
        BUG();
   spin_unlock(&kmap_lock);
   return (void*) vaddr;
}
```

이 함수는 kmap_lock스핀잠그기를 획득하여 다중처리기체계에서 페지표를 동시에 접근하지 않도록 보호한다. kmap()함수를 새치기처리기나 미룰수 있는 함수에서 호출할수 없기때문에 새치기를 금지할 필요는 없다. 다음으로 kmap_high()함수는 페지서술자의 virtual마당이 NULL이 아닌 선형주소를 포함하는지 검사한다.

NULL이라면 map_new_vittual()함수를 호출하여 pkmap_page_table에 있는 입

구점에 페지들의 물리주소를 삽입한다. 다음으로 kmap_high()함수는 다른 핵심부구성요소가 해당 페지들을 사용할수 있으므로 페지들의 선형주소에 해당하는 계수기를 하나 증가시킨다. 마지막으로 kmap_lock스핀잠그기를 해제하고 해당 페지를 배치하는 선형주소를 반환한다.

map_new_virtual()함수는 기본적으로 중첩된 순환 두개를 실행한다.

```
start:
   count = LAST_PKMAP;
   for (;;) {
           last_pkmap_nr = (last_pkmap_nr + 1) & LAST_PKMAP_MASK;
           if (!last_pkmap_nr) {
                   flush_all_zero_pkmaps();
                   count = LAST_PKMAP;
           }
           if (!pkmap_count[last_pkmap_nr])
                   break;
           if (--count)
                   continue;
           {
                   DECLARE_WAITQUEUE(wait, current);
                   __set_current_state(TASK_UNINTERRUPTIBLE);
                   add_wait_queue(&pkmap_map_wait, &wait);
                   spin_unlock(&kmap_lock);
                   schedule();
                   remove_wait_queue(&pkmap_map_wait, &wait);
                   spin_lock(&kmap_lock);
                   if (page_address(page))
                           return (unsigned long)page_address(page);
                   goto start;
           }
           }
```

안쪽 순환에서는 pkmap_count에 있는 모든 계수기를 검색하여 0인 값을 찾는다. last_pkmap_nr변수는 pkmap_page_table페지표에서 마지막으로 사용한 입구점의 색인을 보관한다. 따라서 이전에 map_new_virtual()함수를 호출했을 때 빠져나갔던 위치

부터 검색을 시작한다.

pkmap_count에 있는 마지막 계수기에 도달하면 색인 0에 있는 계수기부터 검색을 다시 한다. map_new_virtual()함수는 검색을 계속하기 전에 flush_aIl_zero_pkmaps()함수를 호출한다. 이 함수는 값이 1인 계수기를 찾는 다른 검색을 시작한다. 계수기값이 1이면 pkmap_page_tabla있는 해당 입구점을 사용하지 않고있지만 해당 TLB 입구점을 비우지 않아서 이것을 사용할수 없음을 의미한다. 그래서 flush_all_zero_pkmaps()를 호출하여 이런 입구점에 대한 TLB를 비우고 계수기를 0으로 만든다.

안쪽순환이 pkmap_count에서 값이 0인 계수기를 찾지 못하면 map_new_virtual() 함수는 다른 프로쎄스가 pkmap_page_table 페지표에 있는 입구점을 해제할 때까지 현재프로쎄스를 차단한다. 이것은 current를 pkmap_map_wait대기렬에 삽입하고 current의 상태를 TASK_UNINTERRUPTIBLE로 설정한 후 schedule() 함수를 호출하여 CPU를 반납함으로써 이루어진다. 일단 프로쎄스가 깨여나면 이 함수는 페지서술자의 virtual마당을 보고 다른 프로쎄스가 이미 해당 페지를 배치했는가를 검사한다. 아직 다른 어떤 프로쎄스도 페지를 배치하지 않았다면 안쪽순환을 다시 시작한다.

안쪽순환에서 0인 계수기를 발견하면 map_new_virtual()함수는 다음을 실행한다.

1.계수기에 해당하는 선형주소를 계산한다.

2.페지의 물리주소를 pkmap_page_table에 있는 입구점에 기록한다. 또한 같은 입구점의 Accessed(접근), Dirty(불결), Read/Wlite(읽기/쓰기), Present(존재)비트를 설정한다.

3.pkmap_count계수기를 1로 설정한다.

4.선형주소를 페지서술자의 virtual마당에 기록한다.

5.선형주소를 되돌린다.

kunmap()함수는 영구핵심부배치을 제거한다. 이 함수는 페지가 실제로 웃자리기억기령역에 있으면 kunmap_high() 함수를 호출한다. 이 함수는 기본적으로 다음코드와 동등하다.

```
void fastcall kunmap_high(struct page *page)
{
   unsigned long vaddr;
   unsigned long nr;
   int need_wakeup;

   spin_lock(&kmap_lock);
   vaddr = (unsigned long)page_address(page);
```

```
    if (!vaddr)
            BUG();
    nr = PKMAP_NR(vaddr);

    need_wakeup = 0;
    switch (--pkmap_count[nr]) {
    case 0:
            BUG();
    case 1:
            need_wakeup = waitqueue_active(&pkmap_map_wait);
    }
    spin_unlock(&kmap_lock);

    if (need_wakeup)
            wake_up(&pkmap_map_wait);
}
```

페지표입구점의 계수기가 1(사용하지 않음)이 되면 kunmap_high()는 pkmap_map_wait대기렬에서 대기중인 프로쎄스를 깨운다.

◦ **림시핵심부배치**

림시핵심부배치의 구현은 영구핵심부배치보다 간단하다. 게다가 현재프로쎄스를 차단하는 일이 결코 없으므로 새치기처리기와 미룰수 있는 함수에서도 사용할수 있다.

웃자리기억기에 있는 어떤 페지틀이든 핵심부주소공간에 있는 창 즉 이 목적을 위해 예약해놓은 페지표입구점을 통해 배치할수 있다. 림시핵심부배치용으로 예약한 창의 수는 매우 적다.

```
enum km_type {
    KM_BOUNCE_READ,
    KM_VSTACK_BASE,
    KM_VSTACK_TOP = KM_VSTACK_BASE + STACK_PAGE_COUNT-1,

    KM_LDT_PAGE15,
    KM_LDT_PAGE0 = KM_LDT_PAGE15 + 16-1,
    KM_USER_COPY,
    KM_VSTACK_HOLE,
    KM_SKB_SUNRPC_DATA,
```

```
    KM_SKB_DATA_SOFTIRQ,
    KM_USER0,
    KM_USER1,
    KM_BIO_SRC_IRQ,
    KM_BIO_DST_IRQ,
    KM_PTE0,
    KM_PTE1,
    KM_IRQ0,
    KM_IRQ1,
    KM_SOFTIRQ0,
    KM_SOFTIRQ1,
    KM_CRASHDUMP,
    KM_UNUSED,
    KM_TYPE_NR
};
```

핵심부는 두 핵심부조종경로가 동시에 같은 창을 사용하지 않게 해야 한다. 따라서 해당 창을 사용할수 있도록 허가한 핵심부구성요소의 이름을 따라 기호의 이름을 지었다. 마지막기호인 KM_TYPE_NR은 선형주소가 아닌 모든 CPU가 사용할수 있는 서로 다른 창의 수를 나타낸다.

마지막을 제외한 km_type에 있는 매 기호는 고정배치하는 선형주소의 색인이다(《 고정배치하는 선형주소 》를 참고). enum fixed_addresses자료구조체는 FIX_KMAP_BEGIN과 FIX_KMAP_END기호도 정의한다. FIX_KMAP_END는 FIX_KMAP_BEGIN + (KM_TYPE_NR*NR_CPUS) - 1값을 가진다. 이런 식으로 체계에 있는 각 CPU마다 고정배치하는 선형주소가 KM_TYPE_NR개 있다. 핵심부는 kmap_pte변수를 fix_to_virt(FIX_KMAP_BEGIN)선형주소에 해당하는 페지표입구점의 주소로 초기화한다.

림시핵심부배치을 만들 때에는 kmap_atomic()함수를 호출한다. 이 함수는 기본적으로 다음코드와 같다.

```
void *kmap_atomic(struct page *page, enum km_type type)
{
    enum fixed_addresses idx;
    unsigned long vaddr;

    inc_preempt_count();
```

```
    if (page < highmem_start_page)
        return page_address(page);

    idx = type + KM_TYPE_NR*smp_processor_id();
    vaddr = __fix_to_virt(FIX_KMAP_BEGIN + idx);
#ifdef CONFIG_DEBUG_HIGHMEM
    if (!pte_none(*(kmap_pte-idx)))
        BUG();
#endif
    if (PageReserved(page))
        set_pte(kmap_pte-idx, mk_pte(page, kmap_prot_nocache));
    else
        set_pte(kmap_pte-idx, mk_pte(page, kmap_prot));
    __flush_tlb_one(vaddr);

    return (void*) vaddr;
}
```

type파라메터와 CPU식별자는 요청한 페지를 배치하기 위해 어떤 고정배치하는 선형주소를 사용하겠는가를 지정한다. 이 함수는 해당 페지틀이 웃자리기억기에 속하지 않으면 페지틀의 선형주소를 반환한다. 웃자리기억기에 속하면 고정배치하는 선형주소에 해당하는 페지표입구점을 해당 페지의 물리주소와 함께 Present(존재), Accessed(접근), Read/Wnte(읽기/쓰기), Dirty(불결)비트로 설정한다. 마지막으로 선형주소에 해당하는 TLB입구점을 비운다.

림시 핵심부배치를 해제할 때에는 kunmap_atomic()함수를 사용한다. 80x86기본방식에서 이 함수는 아무일도 하지 않는다.

림시핵심부배치는 조심해서 사용해야 한다. 림시핵심부배치을 사용하는 핵심부조종경로는 차단되여서는 안된다. 차단되면 다른 핵심부조종경로가 같은 창을 사용하여 다른 웃자리기억기페지를 배치할수 있기때문이다.

7) 형제체계알고리듬

핵심부는 련속적인 페지틀그룹을 할당하는 견고하고 효률적인 방책을 세워야 한다. 이때 기억기관리와 관련한 유명한 문제인 외부단편화(external fragmentation)를 해결해야 한다. 외부단편화는 다른 크기의 련속적인 페지틀그룹을 빈번하게 할당하고 해제하여 할당한 페지틀블로크사이에 작은 여유페지틀 여러개가 《산재》하는 현상이다. 그 결과 후에는 큰 크기의 련속된 페지틀할당을 요청할 때 이것을 담을 충분한 여유페지가 있

어도 기억기를 할당하지 못할수 있다.

이러한 외부단편화를 피할수 있는 방법은 기본적으로 두가지이다.

- 페지화회로를 리용하여 불련속적인 여유페지들의 그룹을 련속된 선형주소구간에 배치한다.
- 남아있는 련속된 여유페지틀블로크를 관리하는 적절한 기법을 개발하여 작은 블로크를 요청할 때 큰 여유블로크를 쪼개서 할당하는 경우를 가능한 줄인다.

핵심부는 다음 3가지 리유때문에 두번째 접근법을 요구한다.

- 련속된 선형주소로는 요청을 처리할수 없고 실제로 련속된 페지틀이 필요한 경우가 종종 있다. 전형적인 실례는 DMA처리기에 할당할 완충용기억기를 요청하는것이다. 입출력연산 한번으로 여러 디스크분구를 전송할 때 DMA는 페지화회로를 무시하고 주소모선에 직접 접근하므로 요청한 완충기는 련속된 페지틀에 있어야 한다.
- 련속된 페지틀이 꼭 필요한 경우가 아니라도 련속된 페지틀을 할당하면 핵심부 페지표을 수정하지 않아도 되므로 큰 리득이 된다. 앞에서 본바와 같이 페지표을 빈번하게 수정하면 CPU가 변환참조완충기의 내용을 비워야 하므로 평균기억기접근시간이 늘어난다.
- 핵심부는 4MB크기의 페지를 활용하여 매우 큰 련속인 물리기억기에 접근할수 있다. 이것은 4kB페지를 사용할 때와 비교하면 변환참조완충기실패(miss)가 줄어들어 평균기억기접근시간을 현저히 향상시켜 준다.(《변환참조완충기》참고)

Linux는 외부단편화문제를 해결하기 위해 유명한 형제체계(buddy system)알고리듬에 기초한 기법을 채택하였다. 사용하지 않는 모든 페지틀을 그룹별로 묶어서 블로크목록 10개에 넣는다. 각 목록은 련속된 페지틀 1, 2, 4, 8, 16, 32, 64, 128, 256, 512개로 구성된 그룹을 담는다. 블로크의 첫번째 페지틀의 물리적인 주소는 그룹크기의 배수이다. 례를 들어 16-페지틀의 시작주소는 16×2^{12}의 배수이다(2^{12}=4096으로 정규페지크기이다).

간단한 실례를 통해 이 알고리듬이 어떻게 동작하는가에 대하여 보기로 하자.

누군가 련속된 페지틀 128개로 구성된 그룹(즉 512kB)을 요청하였다고 하자. 이 알고리듬은 먼저 128-페지틀목록에 여유블로크가 있는가를 검사한다. 여유블로크가 없으면 다음으로 큰 블로크 즉 256-페지틀목록에서 여유블로크를 찾는다. 여기서 여유블로크를 발견하면 페지틀 256개중에서 128개를 요청한쪽으로 할당하고 나머지 페지틀 128개를 여유128페지틀블로크목록에 추가한다. 여유256-페지블로크목록에 없다면 다음으로 큰 블로크, 즉 512-페지틀에서 블로크를 찾는다. 여기서 블로크를 찾으면 페지틀 512개중 128개를 요청한 쪽으로 할당하고 나머지 384개중 처음 256개를 여유 256-페지틀블로크목록에, 남은 페지틀 128개를 여유128-페지틀블로크의 목록에 넣는다. 512페지틀블로크목록이 텅비여있다면 할당을 포기하고 여러 상태를 알려준다.

반대작업인 페지틀블로크를 해제하는데서 이 알고리듬의 이름이 유래되였다. 핵심부

는 크기가 b인 이웃한(형제) 여유블로크쌍을 크기가 2b인 더 큰 블로크 하나로 만들려고 한다. 다음 조건을 만족하면 두 블로크는 형제블로크이다.

· 두 블로크의 크기가 똑같이 b이다.

· 두 블로크가 련속된 물리적인 주소에 위치한다.

· 첫번째 블로크의 첫번째 페지틀의 물리주소는 $2 \times b \times 2^{12}$의 배수이다.

이 알고리듬은 순환적이다. 핵심부는 해제된 블로크를 합친 후에 b의 값을 두배로 늘여 더 큰 블로크를 만들려고 한다.

○ **자료구조**

Linux는 매 령역마다 서로 다른 형제체계를 사용한다. 따라서 80x86기본방식에서는 3가지 형제체계가 있다. 첫째는 ISA_DMA용으로 적합한 페지틀을 다루며 둘째는 보통페지틀을, 셋째는 웃자리기억기페지틀을 다룬다. 각 형제체계는 다음과 같은 핵심자료구조체에 기초한다.

· 앞서 소개한 mem_map배렬, 실제로 매 령역은 mem_map에 있는 항목의 일부분과 관련이 있다. 령역서술자의 zone_mem_map과 size마당은 각각 이 일부분의 첫번째 항목과 항목의 개수를 지정한다.

· 형이 free_area인 요소 10개를 포함하는 배렬, 각 그룹크기마다 항목이 하나씩 있다. 령역서술자의 free_area마당은 이 배렬을 보관한다.

· 비트배치(bitmap)라는 이전 배렬 10개, 매 그룹크기마다 배렬이 하나씩 있다. 각 형제체계는 자기만의 비트배치집합이 있으며 이것으로 자기가 어떤 블로크를 할당했는가를 관리한다.

다음과 같이 free_area자료구조체를 정의한다.

```
struct free_area {
                struct list_head free_ list;
                unsigned long *map;
        }
```

령역서술자에 있는 free_area배렬의 k번째 요소는 크기가 2^k인 블로크의 원형2중련결목록이다. 이 목록의 각 원소는 각 블로크의 첫번째 페지틀의 서술자이다. 페지서술자에 있는 list마당을 통해 이 목록을 구현한다.

map마당은 비트배치에 대한 지적자이다. 비트배치의 크기는 해당 령역에 존재하는 페지틀의 수에 따라 달라진다. free_area배렬의 k번째 입구점의 비트배치에 있는 매 비트는 크기가 2^k인 페지틀의 형제블로크상태 두개를 나타낸다. 비트배치의 비트가 0이라면 쌍을 이룬 두 형제블로크가 모두 사용가능하거나 모두 사용중임을 나타내며 비트가 1이라면 한 블로크만 사용중이라는 사실을 의미한다. 두 형제가 모두 사용가능하면 핵심부는 이것을 크기가 2^{k+1}인 여유블로크 하나로 간주한다.

리해를 위해 128MB의 주기억을 포함하는 령역을 생각해보자. 128MB는 1페지

32768개 또는 2페지그룹16384개, 4페지그룹 8192개, 512페지그룹 64개 등으로 쪼갤 수 있다. 따라서 free_area[0]에 해당하는 비트배치는 페지를 32768개의 매 쌍마다 한 비트씩 16384개 비트로 이루어진다. free_area[1]에 해당하는 비트배치는 두 련속하는 페지를블로크의 각 쌍마다 한 비트씩 8192개 비트로 이루어진다. free_area[9]에 해당하는 마지막 비트배치는 련속된 페지를 512개로 구성된 블로크의 각 쌍마다 한 비트씩 32개 비트로 이루어진다.

그림 4-14는 형제체계알고리듬에서 자료구조를 사용하는 실레이다. zone_mem_map배렬은 우에서 블로크그룹(단일 페지를)하나와 아래에 블로크 4개의 그룹 두개, 이렇게 여유 페지를 아홉개를 포함한다. 량쪽 화살표free_list마당는 구현한 원형2중련결목록를 나타낸다. 비트배치를 비률에 맞게 그리지 않았으므로 주의하시오.

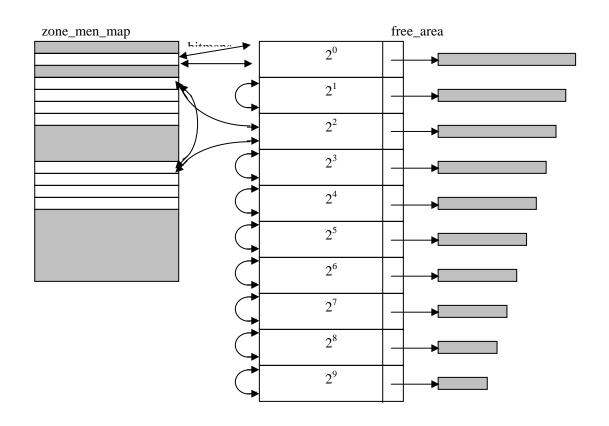



그림 4-14. 형제체계에서 사용하는 자료구조

○ **블로크할당**

alIoc_pages() 함수는 형제체계할당알고리듬의 핵심이다.앞서 살펴본 《페지를 요청과 해제》에서 설명한 다른 모든 할당함수와 마크로는 최종적으로 alloc_pages() 함수를 호출한다.

이 함수는 파라메터 gfp_mask에 들어있는 령역변경자에 해당하는 contig_page_datanode_zonelitsts 배렬에 있는 목록를 조사한다. 이 목록에 있는 첫번째 령역서술자부터 시작하여 해당 령역에 있는 여유페지들의 수(령역서술자의 free_pages 마당에 들어있다.)와 요청한 페지들의 수(alloc_pages()의 order 파라메터), 령역서술자의 pages_1ow 마당에 들어있는 림계값을 비교한다. free_page - 2^{order}가 pages_1ow보다 작거나 같으면 그 령역을 건너 뛰고 목록에 있는 다음령역을 조사한다. 모든 령역에서 여유페지들이 충분하지 않으면 alloc_pages()는 순환을 다시 시작해서 이번에는 적어도 여유페지를 pages_min개를 포함한 령역을 찾는다. 이런 령역도 존재하지 않고 현재프로쎄스가 더 기다릴수 있다면 try_to_free_pages()를 호출하여 기억기요청을 처리하는데 충분한 페지들을 회수한다.(《페지틀회수》참고)

alIoc_pages()는 적합한 여유페지틀수를 가진 령역을 발견하면 buffered_rmqueue()함수를 호출하여 해당 령역에서 블로크를 할당한다. 이 함수는 두 파라메터를 받는다. 하나는 령역서술자의 주소이고 다른 하나는 요청한 여유페지블로크 크기에 로그를 취한 값인 order이다.(0이면 한 페지블로크, 1이면 두 페지블로크, 2이면 네 페지블로크 등)페지틀를 성공적으로 할당하면 buffered_rmqueue() 함수는 할당한 첫번째 페지틀의 페지서술자의 주소를 반환한다. 이 주소는 al1oc_pages()가 반환하는 주소이다. 실패하면buffered_rmqueue()는 NULL을 반환하고 alloc_pages()는 목록에 있는 다음령역을 조사한다.

buffered_rmqueue() 함수는 다음 코드와 동등하다. 먼저 국부변수 몇개를 정의하고 이것을 초기화한다.

```
unsigned long flags;
   struct page *page = NULL;
   int cold = !!(gfp_flags & __GFP_COLD);
```

이 함수는 새치기를 금지하고 후에 블로크를 할당하려고 령역서술자의 마당을 수정할것이므로 해당 령역서술자의 스핀잠그기를 획득한다. 다음으로 요청한 order에 맞는 목록부터 시작해서 필요하면 다음으로 큰 블로크로 계속 진행하여 순환하면서 각 목록에서 사용할수 있는 블로크가 있는지 찾는다.(입구점이 자기를 가리키지 않는것으로 알수 있다.) 이렇게 반복하는 코드는 다음과 같다.

```
if (order == 0) {
```

```
            struct per_cpu_pages *pcp;

            pcp = &zone->pageset[get_cpu()].pcp[cold];
            local_irq_save(flags);
            if (pcp->count <= pcp->low)
                    pcp->count += rmqueue_bulk(zone, 0,
                                            pcp->batch, &pcp->list);
            if (pcp->count) {
                    page = list_entry(pcp->list.next, struct page, lru);
                    list_del(&page->lru);
                    pcp->count--;
            }
            local_irq_restore(flags);
            put_cpu();
    }

    if (page == NULL) {
            spin_lock_irqsave(&zone->lock, flags);
            page = __rmqueue(zone, order);
            spin_unlock_irqrestore(&zone->lock, flags);
    }

    if (page != NULL) {
            BUG_ON(bad_range(zone, page));
            mod_page_state_zone(zone, pgalloc, 1 << order);
            prep_new_page(page, order);
            if (order && (gfp_flags & __GFP_COMP))
                    prep_compound_page(page, order);
    }
    return page;
```

순환이 끝나면 적합한 여유블로크를 찾지 못한것이기때문에 buffered_rmqueue()는 NULL 값을 반환한다. 적합한 여유블로크를 찾은 경우에는 목록에서 첫번째 페지틀서술자를 제거하고 해당 비트배치를 갱신한 후 령역서술자의 free_pages 값을 감소시킨다.(《__rmqueue()함수》 참고)

요청한 크기인 order보다 큰 크기인 curr_order 목록에서 블로크를 찾았다면 다음

교육성 교육정보쎈터

에 나오는 while문을 실행한다. 이 코드의 리론적인 원리는 다음과 같다. 2^h 크기의 페지틀요청을 처리하기 위해 2^k 크기의 페지틀블로크를 사용해야 히는 경우(h<k), 뒤부분의 2^h 페지틀을 할당하고 앞부분의 2^k-2^h페지틀을 h부터 k 사이의 색인에 있는 free_area목록에 할당한다.

마지막으로 rmqueue()는 스핀잠그기를 해제하고 return명령을 실행한다.

그 결과 alloc_pages()함수는 할당한 첫번째 페지틀의 페지서술자의 주소를 되돌린다.

○ **블로크해제**

__free_pages_ok()함수는 페지틀을 해제하는 형제체계방책을 구현한다. 이 함수는 다음 두 입력파라메터를 사용한다.

page

해제할 블로크에 들어있는 첫번째 페지틀서술자의 주소

order

블로크크기에 로그를 취한 값

__free_pages_ok()함수는 보통 련이어 발생하는 할당요청에서 사용할수 있도록 페지틀블로크를 형제체계자료구조체에 삽입한다. 여기에 례외가 하나 있다. 현재프로쎄스가 령역사이에 균형을 맞추려고 기억기령역사이로 페지를 이동할 때 이 함수는 페지틀을 해제하지 않고 블로크를 프로쎄스의 특별한 목록에 삽입한다. __free_pages_ok()는 페지틀블로크로 무엇을 할지 결정하려고 프로쎄스의 PF_FREE_PAGES 기발을 검사한다.

이 기발의 값은 프로쎄스가 기억기령역사이의 균형을 맞출 때에만 1이다. 여기서는 current의 PF_FREE_PAGES 기발이 0이라고 가정하자. 따라서 __free_pages_ok()는 블로크를 형제체계자료구조에 삽입한다.

이 함수는 먼저 몇가지 국부변수를 선언하고 초기화한다.

국부변수 page_idx는 해당 령역의 첫번째 페지틀을 기준으로 하여 해제하는 블로크에서 첫번째 페지틀의 상대적인 색인를 저장한다. 국부변수 index는 비트배치에서 블로크에 해당하는 비트의 번호를 보관한다.

첫번째 페지틀의 PG_referenced와 PG_dirty기발을 지우고 령역에 대한 스핀잠그기를 획득하고 새치기를 금지한다.

국부변수 mask는 2^{order}의 2의 보수값을 보관하며 령역에 있는 여유페지틀의 개수를 증가시킬 때 사용한다.

이제 최대 9-order번 순환하는 반복문을 시작하고 한번 돌 때마다 블로크를 이웃하는 블로크와 합치려 한다. 이 함수는 가장 작은크기부터 시작하여 큰 크기로 올라간다. while순환을 움직이는 조건은 다음과 같다.

순환안에서는 반복할 때마다 mask에 있는 비트를 왼쪽으로 밀기(shift)하고 번호가 page_idx인 블로크과 이웃하는 블로크인 형제블로크를 검사한다.

형제블로크를 사용중이면 순환을 빠져나가지만 형제블로크를 사용중이지 않으면 블

로크를 해당 여유블로크의 목록에서 떼여낸다. 형제블로크의 블로크번호는 page_idx에서 비트 하나를 반대로 바꾸어서 알아낸다.

각 반복이 끝날 때마다 mask와 area, index, page_idx 국부변수를 갱신한다.

그런후 다음 반복을 계속하여 이전 반복에서 다루던것 보다 두배로 큰 여유블로크를 합치려고 시도한다. 이렇게 얻은 여유블로크를 더는 다른 여유블로크와 합칠수 없을 때 반복이 끝난다. 다음으로 이 블로크를 옳바른 목록에 추가한다.

마지막으로 령역에 대한 스핀잠그기를 해제하고 완료한다.

2. 기억기령역관리

여기서는 《기억기령역(memory area)》 즉 련속하는 물리주소를 가진 임의의 길이의 일련의 기억세포(memory cell)를 취급한다.

형제체계알고리듬은 페지틀을 기본기억기령역으로 채택한다. 이것은 상대적으로 큰 기억기요청을 다룰 때에는 좋지만 몇십B나 몇백B처럼 작은 기억기령역의 요청을 어떻게 처리할수 있는가?

말할 필요없이 단지 몇B를 보관하기 위해 한 페지틀 전체를 할당하는것은 매우 경제적인것이 못된다. 이 보다는 한 페지틀내에서 작은 기억기령역을 어떻게 할당하고있는지를 나타내는 새로운 자료구조체를 도입하는것이 더 좋은 접근법이다. 이 과정에서 내부단편화(internal fragmentation)라는 새로운 문제가 발생한다. 이 문제는 요청한 기억기의 크기와 해당 요청을 처리하기 위해 할당하는 기억기령역의 크기가 일치하지 않아서 일어난다.

Linux 초기판본에서 채택한 고전적인 해결책은 기하학적으로 분포된 크기(즉 저장할 자료의 크기가 아닌 2의 거듭 제곱에 따라 정한 크기)의 기억기령역을 제공하는것이다. 이렇게 해서 요청한 기억기크기에 상관없이 내부단편화는 항상 50%가 안됨을 보장할수 있다. 이런 접근법에 따라 핵심부는 크기가 32B부터 131056B까지 기하학적으로 분포된 여유기억기령역목록 13개를 만든다. 새로운 기억기령역을 저장하는데 필요한 페지틀을 추가로 할당하거나 반대로 더는 기억기령역을 포함하지 않은 페지틀을 해제할 때 모두 형제체계을 사용한다. 각 페지틀에 들어있는 여유기억기령역을 관리하기 위해 동적목록를 사용한다.

1) 스랩할당자

형제알고리듬우에서 기억기령역할당알고리듬을 실행하는것은 효률적이지 않다. 더 나은 알고리듬은 1994년 Sun Microsystems가 Solaris 2.4조작체계용으로 개발한 《스랩할당자(slab allocator)》라는 방책에서 유래한다. 이것은 다음과 같은 전제에 바탕을 둔다.

- 저장할 자료의 형이 기억기령역을 할당하는 방법에 영향을 미칠수 있다. 례를들어 사용자방식프로쎄스에 페지틀을 할당할 때 핵심부는 페지를 할당한 후 이것을 0으로

채우는 get_zefoed_page() 함수를 호출한다. 스랩할당자의 개념은 이런 생각을 확장하여 기억기령역을 일련의 자료구조와 《구축자(constructor)와 해제자(destructor)》라는 메쏘드(method), 즉 두 함수를 포함한 객체(oblect)로 바라본다. 구축자는 기억기령역을 초기화하고 해제자는 뒤정리를 한다. 스랩할당자는 객체를 반복해서 초기화하지 않도록 할당하였다가 해제한 객체를 페기하지 않고 기억기에 그대로 저장한다. 새로운 객체를 요청하면 초기화를 다시 하지 않고 기억기에서 이 객체를 가져올수 있다. 실제로 Linux에서 다루는 기억기령역은 초기화나 뒤정리를 할 필요가 없다. 효률성때문에 Linux는 구축자나 해제자메쏘드가 필요한 객체를 사용하지 않는다. 스랩할당자를 도입한 중요한 동기는 형제체계할당자를 호출하는 회수를 줄이는것이다. 따라서 핵심부가 구축자와 해제자메쏘드를 완벽하게 지원하지만 두 메쏘드를 가리키는 지적자는 NULL이다.

- 핵심부함수는 같은 형의 기억기령역을 반복해서 요청하는 경향이 있다. 례를 들어 핵심부는 새로운 프로쎄스를 만들 때마다 프로쎄스서술자나 열린 파일객체 같은 몇가지 크기가 고정된 표용으로 기억기령역을 할당한다. 프로쎄스가 완료하면 이런 표을 포함하든 기억기령역을 재활용할수 있다. 프로쎄스는 매우 자주 만들어지고 없어지므로 스랩할당자가 없다면 같은 기억기령역을 포함하는 페지를을 반복해서 할당하고 해제하느라 시간을 허비한다. 스랩할당자는 이것을 캐쉬에 보관하고 빠르게 재활용 할수 있게 한다.
- 기억기령역에 대한 요청은 그 빈도에 따라 분류할수 있다. 자주 발생할것 같은 특정크기에 대한 요청은 크기가 동일한 특수목적의 객체집단을 만들어 가장 효률적으로 처리함과 동시에 내부단편화 문제도 피할수 있다. 반면에 드물게 나오는 크기는 비록 내부단편화를 일으킬수 있지만 기하학적으로 분포된 크기(이전 Linux에서 사용한 2의 거듭 제곱 같은)인 일련의 객체에 기초한 할당방책을 통해 처리할수 있다.
- 이외에도 기하학적으로 분포된 크기를 사용하지 않는 객체를 도입하여 얻을수 있는 리득이 작다. 자료구조의 시작주소가 2의 거듭 제곱인 물리주소로 집중되는 경향이 덜하므로 처리기의 하드웨어캐쉬성능이 더 좋아진다.
- 형제체계할당자를 호출하는 회수를 가능한 줄이는 일은 하드웨어개쉬의 성능을 위해서도 중요하다. 형제체계함수를 호출할 때마다 하드웨어캐쉬가 불결해지기때문에 평균기억기접근시간이 늘어난다. 핵심부함수하나가 하드웨어캐쉬에 미치는 영향을 《발자취(footprint)》라고 부른다. 이것은 함수가 완료했을 때 해당 함수가 덮어쓴 캐쉬의 백분률로 정의한다. 응당 이 값이 클수록 하드웨어캐쉬는 쓰지 않는 정보로 채워지므로 핵심부함수에서 돌아온 직후에 코드의 실행이 느려진다.

스랩할당자는 객체를 모아서 캐쉬(cache)로 만든다. 각 캐쉬는 형이 같은 객체의 《창고 (store)》이다. 례를 들어 파일을 열면 해당 열린파일(open file)객체를 저장하는데 필요한 기억기령역을 filp(file pointer, 파일지적자)라는 스랩할당자캐쉬에서 가져온다. Linux가 사용하는 스랩할당자캐쉬는 실행중에 /proc/slabinfo파일에서 볼수 있다.

캐쉬가 들어있는 주기억기령역을 스랩으로 나눈다. 각 스랩은 련속된 페지를 하나이상으로 구성되며 할당한 객체와 여유객체를 모두 포함한다.

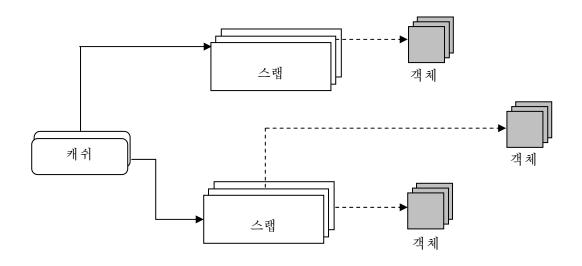



그림 4-15. 스랩할당자의 구성요소

스랩할당자는 빈 스랩의 페지들을 절대로 스스로 해제하지 않는다. 언제 여유기억기가 필요한지 모르고 새로운 객체를 만들수 있는 기억기가 충분히 있을 때 객체를 해제하는것은 아무런 리득이 없다. 따라서 핵심부에 추가로 여유페지들이 필요한 때에만 스랩을 해제한다.

2) 캐쉬서술자

각 캐쉬는 형이 struct kmem_cache_s(kmem_cache_t형과 같다)인 표로 나타난다. 이 표에서 가장 중요한 마당은 다음과 같다.

name

캐쉬이름을 저장하는 문자의 배렬

slabs_full

여유객체가 없는 스랩서술자의 원형2중련결목록

slabs_partial

여유객체와 사용중인 객체를 모두 포함하는 스랩서술자의 원형2중련결목록

slabs_free

여유객체만 포함하는 스랩서술자의 원형2중련결목록

spinlock

다중처리기체계에서 캐쉬에 동시에 접근하지 않도록 보호하는 스핀잠그기

num

스랩 하나에 들어있는 객체의 수(캐쉬의 모든 스랩은 크기가 같다.)

교육성 교육정보쎈터

objsize

캐쉬에 들어 있는 객체의 크기(기억기에서 객체의 시작주소를 정렬해야 하면 스랩할당자는 이 크기를 늘일수 있다)

gfporder

스랩 하나에 들어가는 련속된 페지틀수에 로그를 취한 값

ctor, dtor

각각 캐쉬객체와 관련된 구축자와 해제자 메쏘드를 가리킨다. 앞서 설명한것처럼 지금은 NULL로 설정한다.

next

캐쉬서술자의 2중련결목록을 가리킨다.

flags

캐쉬의 영구적인 속성을 나타내는 기발 집합. 례를 들어 기억기에 객체서술자를 저장하는 두가지 방법중 어느것을 사용하는지를 나타내는 기발이 있다.

dflags

캐쉬의 동적인 속성을 나타내는 기발 집합. 례를 들어 핵심부가 캐쉬에 새 스랩을 할당하고있는지 여부를 나타내는 기발이 있다. 다중처리기체계에서는 캐쉬의 스핀잠그기를 획득한 후 이 기발에 접근해야 한다.

gfpflags

페지틀을 할당할 때 형제체계에 전달하는 기발집합. 대체로 이 마당은 __GFP_DMA의 __GFP_HIGHMEM 령역변경자를 지정한다. 례를 들어 gfpflags에서 __GFP_DMA를 설정하면 캐쉬에 있는 객체를 LSA DMA을 사용할수 있다.

3) 스랩서술자

캐쉬의 각 스랩은 형이 struct slab인 자기만의 서술자가 있다. 스랩서술자를 저장할수 있는 장소는 두가지이다.

- **외부스랩서술자**

  스랩외부에 cache_sizes가 가리키는 ISA DMA용이 아닌 일반 캐쉬중 하나에 보관한다.

- **내부스랩서술자**

  스랩내부에 스랩에 할당한 첫번째 페지틀의 시작부분에 저장한다.

스랩할당자는 객체의 크기가 512B보다 작거나 내부단편화에 의해 스랩내에 스랩서술자와 뒤에서 설명할 객체서술자를 담을 충분한 공간이 있으면 후자의 방법을 선택한다.

스랩서술자에서 가장 중요한 마당은 다음과 같다.

- inuse

  스랩에서 현재 할당한 객체의 수이다.

- s_mem

스랩에 있는 첫번째 여유객체를(할당여부에 상관없이) 가리킨다.

- free

스랩에 있는 첫번째 여유객체를(여유객체가 있다면) 가리킨다.

- list

3가지 스랩서술자의 2중련결목록(캐쉬서술자에 있는 slabs_full이나 slabs_partial, slabs_free)중 하나를 가리키는 지적자이다.

그림 4-16는 캐쉬서술자와 스랩서술자사이의 주요관계를 보여준다. 가득찬 스랩과 일부만 찬 스랩, 텅빈 스랩을 서로 다른 목록으로 련결한다.

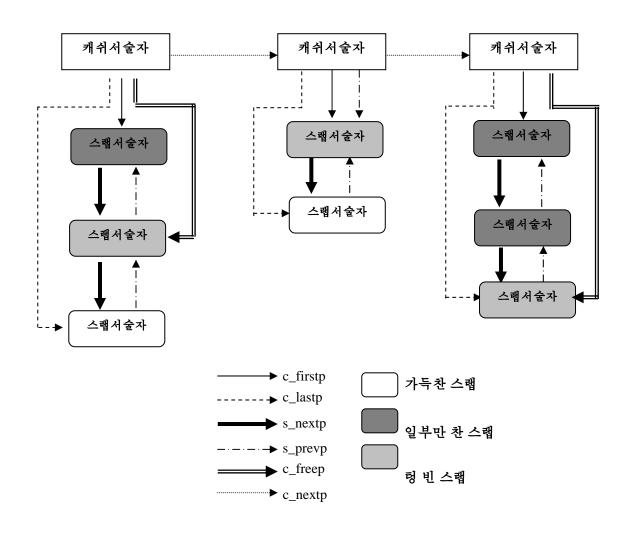



그림 4-16. 캐쉬서술자와 스랩서술자사이의 관계

4) 일반캐쉬와 특수캐쉬

캐쉬는 일반캐쉬와 특수캐쉬 두 종류로 나눈다. 일반캐쉬(general cache)는 스랩할당자가 자기의 필요에 의해서만 사용하는 반면에 특수캐쉬(specific cache)는 핵심부의 나머지부분에서 사용한다.

일반캐쉬의 특징은 다음과 같다.

- 첫번째 캐쉬는 핵심부가 사용하는 나머지캐쉬의 캐쉬서술자를 포함한다. cache_cache변수에 이 서술자가 들어있다.
- 추가캐쉬 26개에는 기하학적으로 분포된 기억기령역이 들어있다.
- cache_sizes 표(cache_sizes_t 형인 요소를 담는다)은 각각 크기가 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536, 130172B인 기억기령역에 관련된 캐쉬서술자 26개를 가리킨다. 각 크기마다 캐쉬가 두개 있다. 하나는 ISA DMA용이며 디른 하나는 일반 할당용이다. 지정한 크기에 해당하는 캐쉬주소를 효률적으로 추출하기 위해 cache_sizes표를 사용한다.

핵심부는 초기화과정에서 kmem_cache_init()과 kmem_cache_sizes_init() 함수를 호출해서 일반캐쉬를 구성한다.

특수캐쉬는 kmem_cache_create() 함수로 만든다. 이 함수는 먼저 파라메터에 따라 새로운 캐쉬를 다룰 가장 좋은 방법을 결정한다.(례를 들어 스랩서술자를 스랩내부에 놓겠는지 아니면 외부에 놓겠는지 여부) 다음으로 cache_cache 일반캐쉬로부터 새 캐쉬용으로 캐쉬서술자를 새로 할당하고 이것을 캐쉬서술자목록(첫번째 요소가 cache_cache이다)에 넣는다.

kmem_cache_desttoy()를 호출하여 캐쉬를 없앨수도 있다. 이 함수는 모듈이 자기를 적재할 때 자기만의 캐쉬를 만들고 부리울 때 캐쉬를 제거하는 경우에 가장 유용하다. 기억기공간을 랑비하지 않도록 핵심부는 캐쉬 자체를 없애기전에 캐쉬에 들어있는 모든 스랩을 없애야 한다. kmem_cache_shrink() 함수는 slab_destroy() 함수를 반복 호출하여 캐쉬에 있는 모든 스랩를 없앤다(뒤에 나오는 《캐쉬에서 스랩해제》참고). 캐쉬서술자의 growing 마당은 어떤 핵심부조종경로가 새로 스랩을 할당하려 할 때 다른 핵심부조종경로가 kmem_cache_shrink()를 호출하여 캐쉬를 없애는 일을 막는다.

실행중에 /proc/slabinfo파일을 읽어서 모든 일반캐쉬와 특수캐쉬의 이름을 알수 있다. 이 파일은 각 캐쉬마다 여유객체의 수와 할당한 객체의 수도 알려준다.

5) 형제체계와 스랩할당자의 련결

스랩할당자가 새 스랩을 만들 때에는 형제체계알고리듬을 사용하여 련속된 여유페지틀을 할당 받는다. 이를 위해 kmem_getpages() 함수를 호출한다.

```
static void *kmem_getpages(kmem_cache_t *cachep, int flags, int
nodeid)
{
```

```
    struct page *page;
    void *addr;
    int i;

    flags |= cachep->gfpflags;
    if (likely(nodeid == -1)) {
            addr = (void*)__get_free_pages(flags,
cachep->gfporder);
            if (!addr)
                    return NULL;
            page = virt_to_page(addr);
    } else {
            page = alloc_pages_node(nodeid, flags,
cachep->gfporder);
            if (!page)
                    return NULL;
            addr = page_address(page);
    }

    i = (1 << cachep->gfporder);
    if (cachep->flags & SLAB_RECLAIM_ACCOUNT)
            atomic_add(i, &slab_reclaim_pages);
    add_page_state(nr_slab, i);
    while (i--) {
            SetPageSlab(page);
            page++;
    }
    return addr;
}
```

각 입구파라메터의 의미는 다음과 같다.

cachep

추가로 페지들이 필요한 캐쉬의 캐쉬서술자를 가리킨다.(필요한 페지들의 수는 cachep->gfporder 마당를 가지고 결정한다.)

flags

페지틀을 할당할 방법을 지정한다.(앞서 살펴본 《페지틀 요청과 해제》 참고) 여기

에 지정한 기발과 캐쉬서술자의 gfpflage에 들어있는 특별한 캐쉬할당기발을 결합한다.

nodeid

마디번호를 가리킨다.

반대작업으로 스랩할당자에 할당한 페지틀을 해제할수 있다.(뒤에 나오는 《캐쉬에서 스랩 해제》참고)

```
static void kmem_freepages(kmem_cache_t *cachep, void *addr)
{
   unsigned long i = (1<<cachep->gfporder);
   struct page *page = virt_to_page(addr);
   const unsigned long nr_freed = i;

   while (i--) {
          if (!TestClearPageSlab(page))
                 BUG();
          page++;
   }
   sub_page_state(nr_slab, nr_freed);
   if (current->reclaim_state)
          current->reclaim_state->reclaimed_slab += nr_freed;
   free_pages((unsigned long)addr, cachep->gfporder);
   if (cachep->flags & SLAB_RECLAIM_ACCOUNT)
          atomic_sub(1<<cachep->gfporder, &slab_reclaim_pages);
}
```

이 함수는 물리주소가 addr인 페지틀부터 시작해서 cachep가 가리키는 캐쉬의 스랩에 할당한 페지틀을 해제한다.

6) 캐쉬에 스랩할당

새로 만든 캐쉬는 이무런 스랩도 포함하지 않으므로 여유객체 또한 포함하지 않는다. 다음 두 조건에 모두 해당하는 경우에만 캐쉬에 새로운 스랩을 할당한다.

- 새 객체를 할당해 달라는 요청을 받을 때
- 캐쉬에 여유 객체가 없을 때

이런 일이 일어나면 스랩할당자는 cache_grow()를 호출하여 캐쉬에 새로 스랩을 할당한다. 이 함수는 kmem_getpages()를 호출해서 형제체계에서 페지틀그룹을 할당받

고 alloc_slabmgmt()를 호출해서 새로운 스랩서술자를 받는다. 새 스랩서술자는 스랩의 첫번째 페지틀이나 cache_sizes가 가리키는 차수가 0인 일반캐쉬내의 스랩에 존재한다. 다음으로 cache_init_objs()를 호출하여 구축자메쏘드가 있으면 새로 만든 스랩에 들어가는 모든 객체에 이 구축자를 적용한다.

다음으로 cache_grow()는 새로운 스랩에 할당한 페지틀의 모든 페지서술자를 검색하여 페지서술자에 있는 list 마당내의 next와 prev 마당을 각각 캐쉬서술자의 주소와 스랩서술자의 주소로 설정한다. 형제체계에 있는 함수는 list 마당을 페지틀이 여유페지틀일 때에만 사용하는데 스랩할당자함수가 다루는 페지틀은 형제체계립장에서는 여유 페지틀이 아니기때문에 이렇게 페지틀 서술자를 다른 용도로 사용하더라도 형제체계는 혼동하지 않는다. 이 함수는 페지틀의 PG_slab기발도 설정한다.

다음으로 스랩서술자 *slabp를 캐쉬서술자 *cachep의 텅빈 스랩목록끝에 추가한다.

7) 캐쉬에서 스랩해제

앞서 언급한대로 스랩할당자는 절대 빈 스랩의 페지틀을 스스로 해제하지 않는다. 실제로 다음 두 조건에 모두 해당할 때에만 스랩을 해제한다.

- 형제체계가 새로 페지틀그룹을 할당해달라는 요청을 처리할수 없을 때(령역에 기억기가 조금만 남은 때)
- 스랩이 비여있을 때 즉 스랩에 들어있는 모든 객체가 사용가능할 때 핵심부는 여유페지틀이 부족하여 추가로 여유페지틀을 찾을 때 try_to_free_pages()를 호출한다. 이 함수는 kmem_cache_reap()을 호출해서 적어도 빈 스랩 하나를 포함하는 캐쉬를 선택한다. 다음으로 slab_destroy()를 호출하여 스랩을 텅빈 스랩목록에서 제거하고 스랩을 없앤다.

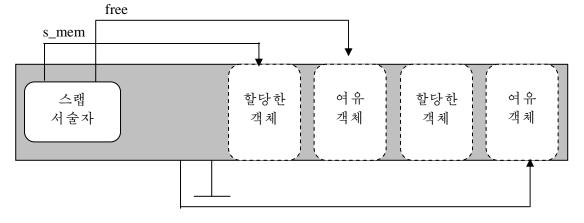
```
static void slab_destroy (kmem_cache_t *cachep, struct slab *slabp)
{
    void *addr = slabp->s_mem - slabp->colouroff;

#if DEBUG
    int i;
    for (i = 0; i < cachep->num; i++) {
            void *objp = slabp->s_mem + cachep->objsize * i;

            if (cachep->flags & SLAB_POISON) {
#ifdef CONFIG_DEBUG_PAGEALLOC
                    if ((cachep->objsize%PAGE_SIZE)==0 && OFF_SLAB(cache
p))
```

```
				kernel_map_pages(virt_to_page(objp), cachep->objsi
ze/PAGE_SIZE,1);
			else
				check_poison_obj(cachep, objp);
#else
			check_poison_obj(cachep, objp);
#endif
		}
		if (cachep->flags & SLAB_RED_ZONE) {
			if (*dbg_redzone1(cachep, objp) != RED_INACTIVE)
				slab_error(cachep, "start of a freed object "
							"was overwritten");
			if (*dbg_redzone2(cachep, objp) != RED_INACTIVE)
				slab_error(cachep, "end of a freed object "
							"was overwritten");
		}
		if (cachep->dtor && !(cachep->flags & SLAB_POISON))
			(cachep->dtor)(objp+obj_dbghead(cachep), cachep, 0);
	}
#else
	if (cachep->dtor) {
		int i;
		for (i = 0; i < cachep->num; i++) {
			void* objp = slabp->s_mem+cachep->objsize*i;
			(cachep->dtor)(objp, cachep, 0);
		}
	}
#endif

	if (unlikely(cachep->flags & SLAB_DESTROY_BY_RCU)) {
		struct slab_rcu *slab_rcu;

		slab_rcu = (struct slab_rcu *) slabp;
		slab_rcu->cachep = cachep;
		slab_rcu->addr = addr;
```

```
            call_rcu(&slab_rcu->head, kmem_rcu_free);
        } else {
            kmem_freepages(cachep, addr);
            if (OFF_SLAB(cachep))
                kmem_cache_free(cachep->slabp_cache, slabp);
        }
    }
```

이 함수는 캐쉬에 객체용 해제자메쏘드가 있는지 검사하여(dtor마당이 NULL이 아닌지) 있는 경우 스랩에 있는 모든 객체에 해제자를 적용한다. 이때 국부변수 objp는 현재 처리중인 객체를 계속해서 유지한다. 다음으로 kmem_freepages()를 호출하여 스랩이 사용하던 모든 련속된 페지들을 형제체계에 되돌린다. 마지막으로 스랩서술자가 스랩외부에 있으면(뒤에서 설명하는데 OFF_SLAB 마크로가 1을 되돌리는 경우이다.) 스랩서술자의 캐쉬에서 이것을 해제한다.

8) **객체서술자**

각 객체는 kmem_bufctl_t 형인 서술자를 포함한다. 객체서술자를 해당 스랩서술자 뒤에 있는 배렬에 저장한다. 따라서 스랩서술자와 마찬가지로 스랩의 객체서술자도 그림 4-17에서 보는바와 같이 두가지 방법으로 저장할수 있다.

외부에 서술자가 있는 스랩

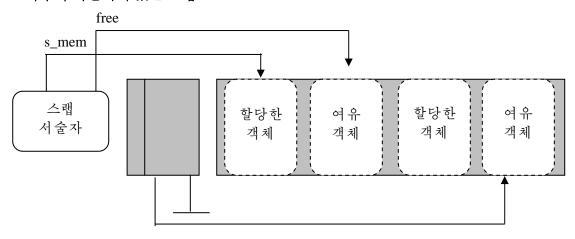



그림 4-17. 스랩서술자와 객체서술자사이의 관계

o 외부객체서술자

스랩외부에 cache_sizes가 가리키는 일반캐쉬중 하나에 저장한다. 따라서 기억기령역의 크기는(따라서 객체서술자를 저장하는데 어떤 일반캐쉬를 사용할지는) 스랩에 저장하는 객체의 수(캐쉬서술자의 num마당)에 따라 달라진다.

o 내부객체서술자

스랩내부에 객체서술자가 서술하는 객체 바로앞에 저장한다.

배렬에 있는 첫번째 객체서술자는 스랩에 있는 첫번째 객체를 서술하며 나머지도 마찬가지이다. 객체서술자는 단순히 부호없는 옹근수이며 객체를 사용하지 않을 때에만 의미가 있다. 이것은 스랩내에서 사용할수 있는 다음 객체의 색인를 지정한다. 따라서 이것을 통해 스랩내에 있는 여유 객체의 단순목록를 구현한다. 여유객체목록에 있는 마지막요소의 객체서술자는 미리 약속한 값인 BUFCTL_END(0xffffffff)로 표시한다.

9) 기억기에서 객체정렬

스랩할당자는 자기가 관리하는 객체를 기억기에서 정렬(align)한다. 즉 시작물리주소가 지정한 상수(보통의 2의 거듭제곱이다)의 배수가 되도록 객체를 기억세포에 보관한다. 이 상수를 《정렬결수(alignment factor)》라고 한다.

스랩할당자에서 사용할수 있는 최대 정렬결수는 4096 즉 폐지틀의 크기이다. 이것은 객체를 객체의 물리주소나 선형주소를 통해 정렬할수 있음을 의미한다. 두 경우 모두 정렬할 때 주소의 아래자리 12bit만 바꿀수 있다.

보통 극소형콤퓨터는 물리주소가 단어(word)크기 즉 콤퓨터의 내부기억기모선의 폭에 정렬되여있을 때 기억세포에 좀 더 빨리 접근한다. 따라서 kmem_cache_create()함수는 기본적으로 객체를 BYTFS_PER_WORD 마크로가 지정한 단어크기에 따라 정렬하려고 한다. Intel펜티움처리기에서는 단어가 32bit길이므로 이 마크로 값은 4이다.

새로 스랩캐쉬를 만들 때 캐쉬에 들어가는 객체를 CPU의 1차 하드웨어캐쉬를 기준으로 정렬하도록 지정할수 있다. 이렇게 하려면 SLAB_HWCACHE_ALIGN 캐쉬서술자기발을 지정해야 한다. kmem_cache_create() 함수는 이 요청을 다음과 같이 처리한다.

- 객체크기가 캐쉬행(cache line)의 절반보다 크면 L1_CACHE_BYTES의 배수가 되도록 즉 캐쉬행 시작에 있도록 객체를 주기억에 정렬한다.

- 그렇지 않으면 객체크기를 L1_CACHE_BYTES 계수값으로 반올림한다. 이것은 한 객체가 절대로 두 캐쉬행에 걸쳐 있지 않도록 만들어준다.

여기서 스랩할당자가 하는 일은 기억기공간과 접근시간사이의 흥정이다. 인공적으로 객체크기를 크게하면 더 좋은 캐쉬성능을 얻을수 있지만 내부단편화를 더 심화시킨다.

10) 스랩채색(coloring)

앞에서는 같은 하드웨어캐쉬행은 주기억의 서로 다른 여러 블로크를 배치한다는것을 취급하였다. 여기서는 크기가 동일한 객체를 캐쉬에서 편위에 저장하는 경향이 있다는 사실도 살펴보았다. 다른 스랩에서 동일한 편위에 있는 객체는 결국 같은 캐쉬행으로 배치될 가능성이 상대적으로 매우 높다. 따라서 캐쉬하드웨어는 주기억의 서로 다른 위치에 존재하는 두 객체를 같은 캐쉬행으로 번갈아 가며 가져오면서 기억기주기을 랑비하고 다른 캐쉬행을 조금밖에 사용하지 않을수도 있다. 스랩할당자는 이런 유쾌하지 않은 캐쉬동작을 《스랩채색(slab coloring)》이라는 방책으로 줄이려 한다. 스랩채색이란 스랩에 색(color)이라는 서로 다른 임의의 값을 할당하는것을 말한다.

스랩채색을 살펴보기에 앞서 캐쉬에서 객체를 어떻게 배치하는지 알아보자. 객체를 주기억이 정렬하고있는 캐쉬를 생각해보자. 즉 객체의 주소는 어떤 정수값 aln의 배수이여야 한다. 정렬제한까지 고려하더라도 객체를 스랩내에 둘수 있는 방법은 여러가지이다. 이중 어떤것을 선택할지는 다음의 변수가 좌우한다.

num

스랩에 저장할수 있는 객체의 수. 이 값은 캐쉬서술자의 num 마당에 들어있다.

osize

정렬바이트를 포함한 객체의 크기

dsize

스랩서술자크기에 모든 객체서술자의 크기를 합한값. 스랩서술자와 객체서술자를 스랩외부에 저장하는 경우 이 값은 0이다.

free

스랩내에서 사용하지 않는 바이트(어떤 객체에도 할당하지 않는 바이트)의 수

스랩의 전체 바이트길이는 다음과 같이 표현할수 있다.

스랩길이= (num × osize) + dsize + free

여기서 free는 항상 osize보다 작다. 그렇지 않으면 스랩내에 객체를 추가로 더 넣을수 있기때문이다. 그러나 free는 aln보다 클수 있다.

스랩할당자는 사용하지 않는 *free*바이트를 사용하여 스랩에 색을 부여한다. 《색(color)》이라는 용어는 단순히 스랩을 세분화하고 기억기할당자가 객체를 다른 선형주소사이에 퍼뜨릴수 있도록 하려고 사용한다. 이렇게 해서 핵심부는 극소형처리기의 하드웨어캐쉬성능을 가능한 최대화한다.

다른 색의 스랩은 정렬제한을 만족하는 범위내에서 스랩의 첫번째 객체를 각기 다른 기억기위치에 저장한다. 사용가능한 색수는 (*free/aln*)+1이다. 첫번째 색은 0, 마지막 색은(이 값은 캐쉬서술자의 color 마당에 들어있다) *free/aln*으로 표현한다.

스랩의 색이 *col*이면 이 스랩의 첫번째 객체의 편위는 (스랩의 시작주소부터 상대적인 주소) *col*×*aln*+*dsize* 바이트이다. 이 값을 캐쉬서술자의 colour_off 마당에 저장한다. 그림4-18은 스랩색에 따라 스랩내부에 객체를 어떻게 배치하는지 보여준다. 채색은 기본적으로 스랩의 빈 령역 일부를 맨 끝에서 맨 앞으로 옮기는것이다.

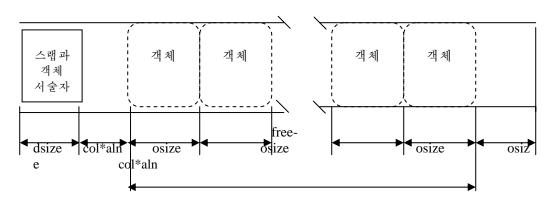



그림 4-18. 색이 col이고 aln으로 정렬하는 스랩

채색은 *free*의 값이 충분히 큰 경우에만 동작한다. 객체를 정렬할 필요가 없거나 스랩내에 사용하지 않는 바이트의 수가 요청한 정렬결수보다 작으면 (*free*<*aln*), 여기서 할수 있는 스랩채색은 스랩이 색 0을 가지는것 즉 첫번째 객체에 편위 0을 할당하는것뿐이다.

현재색을 캐쉬서술자의 colour_next 마당에 저장하여 같은 종류의 객체를 포함하는 스랩사이에서 여러가지 색을 똑같이 분포시킨다. kmem_cache_grow()함수는 colour_next에서 지정하는 색을 새로운 스랩에 할당한 후 이 마당의 값을 증가시킨다. 이 값이 colour값에 도달하면 다시 0으로 만든다. 이런식으로 각 스랩은 이전 색과는 다른 최대 가능한 색개수만큼 서로 다른 색을 가진다.

11) 다중처리기체계에서 객체의 국부배렬

Linux는 다중처리기체계용 스랩할당자를 Solaris2.4와 다르게 구현한다. 처리기들이 스핀잠그기를 다루는 일을 줄이려고 스랩할당자의 각 캐쉬는 체계에 있는 각 CPU마다 작은 수의 여유객체의 지적자배렬을 포함한다. 스랩객체를 할당하고 해제하는것에 해

당하는 수는 국부배렬(local array)에만 영향을 미친다. 국부배렬이 너무 적어지거나 너무 많아진 경우에만 스랩자료구조가 관여한다.

캐쉬서술자는 체계에 있는 CPU마다 하나씩 cpucache_t자료구조체를 가리키는 지적자를 포함하는 cpudata배렬을 포함한다. cpucache_t자료구조체는 객체의 국부배렬의 서술자를 나타내고 다음과 같은 마당를 포함한다.

avail

국부배렬에서 사용할수 있는 객체수: 배렬에서 첫번째 여유슬로트의 색인역활도 한다.

limit

국부배렬의 크기 즉 국부배렬에 있을수 있는 최대객체수

국부배렬서술자는 배렬자체의 주소를 저장하지 않는다. 사실 이 배렬은 서술자 바로 뒤에 있다. 물론 배렬은 캐쉬의 스랩에 들어있는 여유객체에 대한 지적자를 저장하는것이지 객체자체를 저장하는것은 아니다. 배렬의 기본크기는 스랩캐쉬에 저장하고있는 객체의 크기에 따라 달라진다. 작은 객체인 경우(255B까지)배렬크기는 252개이고 중간객체인 경우(256부터1023B사이) 124개, 큰 객체인 경우(1024B이상) 60개이다. 체계관리자는 /proc/slabinfo파일에 쓰기를 하여 각 캐쉬마다 배렬의 크기를 조절할수 있다.

12) 캐쉬에 객체할당

kmem_cache_alloc()함수를 호출하여 새 객체를 얻을수 있다. cachep파라메터는 새로운 여유객체를 가져올 캐쉬서술자를 가리키고 flag파라메터는 형제체계할당자함수에게 전달할 기발을 나타낸다.

이 함수의 구현은 단일처리기체계와 다중처리기체계에서 다르다. 다중처리기체계의 경우 해제한 국부객체배렬도 생각해야 하기때문이다. 이 두 경우를 따로따로 설명한다.

○ 단일처리기인 경우

kmem_cache_alIoc()는 __cache_alloc()를 호출한다. (《__cache_alloc()함수》를 참고)

```
void * kmem_cache_alloc (kmem_cache_t *cachep, int flags)
{
   return __cache_alloc(cachep, flags);
}


   static inline void * __cache_alloc (kmem_cache_t *cachep, int flags)
{
   unsigned long save_flags;
   void* objp;
   struct array_cache *ac;
```

```
    cache_alloc_debugcheck_before(cachep, flags);

    local_irq_save(save_flags);
    ac = ac_data(cachep);
    if (likely(ac->avail)) {
            STATS_INC_ALLOCHIT(cachep);
            ac->touched = 1;
            objp = ac_entry(ac)[--ac->avail];
    } else {
            STATS_INC_ALLOCMISS(cachep);
            objp = cache_alloc_refill(cachep, flags);
    }
    local_irq_restore(save_flags);
    objp = cache_alloc_debugcheck_after(cachep, flags, objp, __builtin_ret
urn_address(0));
    return objp;
}
```

○ **다중처리기인 경우**

kmem_cache_alloc() 함수는 먼저 국부새치기를 금지하고 실행중인 CPU에 관련된 캐쉬의 국부배렬에 여유객체가 있는지 찾는다.

```
1ocal_irq_save(save_flags);
    ac = ac_data(cachep);
            STATS_INC_ALLOCHIT(cachep);
            ac->touched = 1;
            objp = ac_entry(ac)[--ac->avail];
            local_irq_restore(save_flags);
    objp = cache_alloc_debugcheck_after(cachep, flags, objp,
__builtin_return_address(0));
            return objp;
```

13) 캐쉬에서 객체해제

mem_cache_free() 함수는 앞에서 스랩할당자가 할당한 객체를 해제한다. 이 함수는 캐쉬서술자주소인 cachep와 해제할 객체주소인 objp를 파라메터로 받는다. kmem_cach_alloc()와 마찬가지로 단일처리기경우와 다중처리기경우를 구별해서 설명한다.

○ **단일처리기의 경우**

이 함수는 먼저 국부새치기를 금지하고 해당 객체를 포함하는 스랩의 서술자주소를 알아내면서 시작한다. 이 함수는 객체를 저장하는 페지들의 페지서술자가 있는 listprev 마당을 활용한다.

```
slab_t * slabp;
unsigned int objnr;
1ocal_irq_save(save_flags);
slabp=(slab_t *) mem_map[__pa(objp) >> PAGE_SHIFT].list.prev;
```

다음으로 스랩내에 있는 객체의 색인을 계산하고 해당 객체서술자의 주소를 알아내서 객체를 스랩의 여유객체목록의 머리에 추가한다.

```
objnr = (objp - slabp->s_mem) / cachep->objsize;
((kmem bufctl_ t *) (slabp+1)) [objnr] = slabp->free;
slabp->free = objnr;
```

마지막으로 스랩을 다른 목록으로 옮겨야 하는지 검사한다.

```
if (--slabp->inuse == 0) { /* 이제 스랩은 텅빈 상태이다. */
list_del(&slabp->list);
list_add(&slabp->list, &cachep->slabs_free);
} else if (slabp->inuse+1== cachep->num) { /* 스랩이 가득찬 상태였다. */
list_del(&slabp->list);
list_add(&slabp->list, &cachep—>slabs_partial);
}
1ocal_irq_restore(save_flags);
   return;
```

◦ **다중처리기의 경우**

이 함수는 먼저 국부새치기를 금지하면서 시작한다. 다음으로 객체지적자의 국부배렬에 여유슬로트가 있는지 여부를 검사한다.

```
cpucache_t * cc;
1ocal_irq_save(save_flags);
cc=cachep->cpudata[smp_processor_id()];
if (cc->avail== cc->limit) {
cc->avail = cachep->batchcount;
free_block(cachep, &((void *) (cc+1))[cc->avail],
cachep->batchcount);
}
```

```
((void *) (cc+1)) [cc->avail++] = objp;
        1ocal_irq_restore(save_flags);
return;
```

국부배렬에서 여유슬로트가 적어도 하나이상 있다면 여기에 해제하는 객체의 주소를 저장한다. 여유슬로트가 없으면 free_block( )를 호출하여 객체 cachep->batchcount 개를 해제해서 할당자캐쉬로 보낸다. free_block(cachep, objpp, len)함수는 캐쉬스핀잠그기를 해제한 후 objpp주소에 있는 국부배렬입구점부터 객체 len개를 해제한다.

```
  spin_1ock(&cachep->spinlock);
for(; len>0; Ien--, objpp++) {
slab_t * slabp =
(slab_t *)mem_map[__pa(*objpp)>>PAGE_SHIFT].list.prev;
unsigned int objnr=
(*objpp-slabp->s_mem)/cachep->objsize;
((kmem_bufctl_t *) (slabp+1))[objnr]=slabp->free;
slabp->free=objnr;
if (--slabp->inuse == 0) { /* 스랩은 이제 텅빈 상태이다. */
list del(&slabp->list);
list add(&slabp->list, &cachep->slabs_free);
} else if (slabp->inuse+1==cachep->num) { /* 스랩이 가득 찬 상태였다. */
list_del(&slabp->list);
list_add(&slabp->list, &cachep->slabs_partial);
}
}
spin_unlock(&cachep->spinlock);
```

객체를 해제하여 스랩으로 보내는것은 단일처리기경우와 같으므로 더 설명하지 않는다.

14) 범용객체

앞에서 《형제체계알고리듬》에서 설명한것처럼 드물게 일어나는 기억기령역요청은 객체의 크기가 최소 32부터 최대 131072B까지 기하학적으로 분포되여있는 일반캐쉬그룹을 통해 처리한다.

이 종류의 객체는 kmalloc() 함수를 호출해서 얻는다.

```
void * __kmalloc (size_t size, int flags)
{
  struct cache_sizes *csizep = malloc_sizes;
```

```
	for (; csizep->cs_size; csizep++) {
		if (size > csizep->cs_size)
			continue;
#if DEBUG
		BUG_ON(csizep->cs_cachep == NULL);
#endif
		return __cache_alloc(flags & GFP_DMA ?
			csizep->cs_dmacachep : csizep->cs_cachep, flags);
	}
	return NULL;
}
```

이 함수는 cache_sizes표를 사용하여 요청한 크기에 가장 가까운 2의 반복제곱의 크기를 계산한다. 다음으로 _cache_alloc()을 호출해서 객체를 할당한다. __cache_alloc()을 호출할 때 __GFP_DMA기발을 지정했는지 여부에 따라 ISA DMA용 페지들에 대한 캐쉬서술자나 보통 페지들의 캐쉬서술자를 파라메터로 전달한다.

kmalloc()를 호출해서 할당한 객체는 kfree()를 호출하여 해제할수 있다.

```
 void kfree (const void *objp)
{
	kmem_cache_t *c;
	unsigned long flags;

	if (!objp)
		return;
	local_irq_save(flags);
	kfree_debugcheck(objp);
	c = GET_PAGE_CACHE(virt_to_page(objp));
	__cache_free(c, (void*)objp);
	local_irq_restore(flags);
}
```

기억기령역을 포함하는 첫번째 페지틀의 서술자에 있는 list.next 마당을 읽어서 해당 캐쉬서술자를 확인할수 있다. kmem_cache_free()를 호출하여 기억기령역을 해제한다.

3. 불련속적인 기억기령역관리

지금까지 한 론의를 통해 기억기령역을 련속된 페지들의 집합으로 배치하는것이 더 바람직하며 이것을 통해 캐쉬를 더욱 잘 활용할수 있고 평균기억기접근시간을 줄일수 있다는 사실을 알았다. 그러나 기억기령역을 많이 사용하지 않으면 련속된 선형주소로 접근할수 있지만 물리적으로는 불련속적인 페지들을 기반으로 하는 할당방책을 생각해보는것도 리치에 맞는일이다. 이 방책의 가장 큰 우점은 외부단편화(external fragmentation)문제를 피할수 있다는것이다. 반면에 핵심부페지표를 다루어야 하는 부족점도 있다. 응당 불련속인 기억기령역의 크기는 4096의 배수여야 한다. Linux는 몇가지 용도로 불련속적인 기억기령역을 사용한다. 이것을 사용하는 실례로 활성화된 교환령역용으로 자료구조를 할당하거나(《교환령역활성화와 비활성화》 참고)모듈용으로 공간을 할당하거나 일부 입출력구동프로그람용으로 완충기를 할당하는것도 있다.

1) 불련속적인 기억기령역의 선형주소

선형주소의 빈 범위를 찾기 위해 PAGE_OFFSET(보통은 마지막 1GB의 시작주소인 0xc0000000)부터 시작하는 령역을 들여다 볼수 있다. 그림 4-19는 마지막 1GB 선형주소를 어떻게 사용하는지 보여준다.

- 령역의 시작부분에는 주기억의 처음 896MB(《프로쎄스페지표》 참고)를 배치하는 선형주소가 들어간다. high_memory 변수는 직접 배치하는 물리기억기의 끝에 해당하는 선형주소를 저장한다.
- 령역의 끝에는 고정배치하는 선형주소가 들어간다.(《고정배치하는 선형주소》 참고)
- PKMAP_BASE(0*fe000000)부터 핵심부에서 웃자리기억기페지들을 배치하는데 사용하는 선형주소가 있다.(앞에서 본 《웃자리기억기페지들의 핵심부배치》 참고)
- 선형주소의 나머지구간은 불련속적인 기억기령역을 위해 사용할수 있다. 물리적인 기억기의 끝과 첫번째 불련속적인 기억기령역사이에는 안전을 위해 8MB (VMALLOC_OFFSET마크로)의 간격이 있다. 이 간격의 목적은 범위를 벗어난 기억기 접근을 잡아내는것이다. 같은 목적으로 각 불련속적인 기억기령역사이에는 4MB크기의 안전간격이 추가로 들어간다.

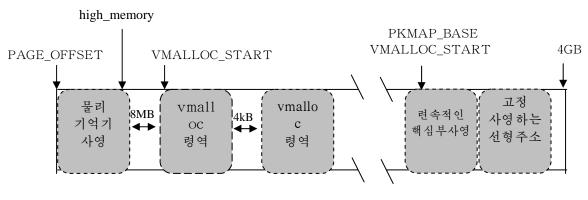



그림 4-19. PAGE_OFFSET 부터 시작하는 선형주소구간

VMALLOC_START마크로는 불련속적인 기억기령역용으로 예약된 선형공간의 시작주소를 정의하고 VMALLOC_END는 그 마지막주소를 정의한다.

2) **불련속적인 기억기령역서술자**

각 불련속적인 기억기령역마다 struct vm_struct 형서술자가 관련된다.

```
struct vm_struct {
    void              *addr;
    unsigned long     size;
    unsigned long     flags;
    struct page       **pages;
    unsigned int      nr_pages;
    unsigned long     phys_addr;
    struct vm_struct  *next;
};
```

이 서술자는 next마당를 리용한 간단한 목록으로 들어간다. vmlist변수는 이 목록에 첫번째 요소의 주소를 보관한다. vmlist_lock 읽기/쓰기스핀잠그기를 사용하여 이 목록에 접근하는것을 보호한다. addr마당에는 기억기령역의 첫번째 기억세포의 선형주소가, size마당에는 령역의 크기에 4096(앞에서 언급한 령역사이의 안전간격크기)을 더한값이 들어간다.

get_vm_area() 함수는 struct vm_struct 형의 새로운 서술자를 만든다. 여기에 넘어가는 파라메터 size는 새로운 기억기령역의 크기를 지정한다. 이 함수는 근본적으로 다음과 같다.

```
struct vm_struct *__get_vm_area(unsigned long size, unsigned long flags,
                        unsigned long start, unsigned long end)
{
    struct vm_struct **p, *tmp, *area;
    unsigned long align = 1;
    unsigned long addr;

    if (flags & VM_IOREMAP) {
            int bit = fls(size);

            if (bit > IOREMAP_MAX_ORDER)
                    bit = IOREMAP_MAX_ORDER;
            else if (bit < PAGE_SHIFT)
                    bit = PAGE_SHIFT;
```

```
        align = 1ul << bit;
    }
    addr = ALIGN(start, align);

    area = kmalloc(sizeof(*area), GFP_KERNEL);
    if (unlikely(!area))
        return NULL;

    size += PAGE_SIZE;
    if (unlikely(!size)) {
        kfree (area);
        return NULL;
    }

    write_lock(&vmlist_lock);
    for (p = &vmlist; (tmp = *p) != NULL ;p = &tmp->next) {
        if ((unsigned long)tmp->addr < addr) {
            if((unsigned long)tmp->addr + tmp->size >= addr)
                addr = ALIGN(tmp->size +
                        (unsigned long)tmp->addr, align);
            continue;
        }
        if ((size + addr) < addr)
            goto out;
        if (size + addr <= (unsigned long)tmp->addr)
            goto found;
        addr = ALIGN(tmp->size + (unsigned long)tmp->addr, align);
        if (addr > end - size)
            goto out;
    }

found:
    area->next = *p;
    *p = area;
```

```
    area->flags = flags;
    area->addr = (void *)addr;
    area->size = size;
    area->pages = NULL;
    area->nr_pages = 0;
    area->phys_addr = 0;
    write_unlock(&vmlist_lock);

    return area;

out:
    write_unlock(&vmlist_lock);
    kfree(area);
    if (printk_ratelimit())
            printk(KERN_WARNING "allocation failed: out of vmalloc spac
e - use vmalloc=<size> to increase size.\n");
    return NULL;
}
```

이 함수는 먼저 kmalloc()을 호출하여 새 서술자를 위한 기억기령역을 할당한다. 다음으로 struct vm_struct형 서술자의 목록를 검색하여 최소 size+4096주소를 포함할수 있는 선형주소구간이 있는가를 찾는다. 이런 구간이 존재하면 서술자에 있는 마당를 초기화하고 불련속적인 기억기령역의 시작주소를 반환하면서 함수를 끝마친다. 그렇지 않고 addr+size가 VMALLOC_END를 넘어서면 get_vm_area()함수는 서술자에 할당했던 기억기를 해제하고 NULL을 반환한다.

3) 불련속적인 기억기령역할당

vmalloc() 함수는 핵심부에 불련속적인 기억기령역을 할당한다. size 파라메터는 요청한 령역의 크기를 나타낸다. 이 함수는 요청한 기억기를 할당할수 있으면 새 령역의 시작선형주소를 반환하고 그렇지 않으면 NULL 지적자를 반환한다.

```
void *vmalloc(unsigned long size)
{
return __vmalloc(size, GFP_KERNEL | __GFP_HIGHMEM,
PAGE_KERNEL);
```

```
}

void *__vmalloc(unsigned long size, int gfp_mask, pgprot_t prot)
{
   struct vm_struct *area;
   struct page **pages;
   unsigned int nr_pages, array_size, i;

   size = PAGE_ALIGN(size);
   if (!size || (size >> PAGE_SHIFT) > num_physpages)
         return NULL;

   area = get_vm_area(size, VM_ALLOC);
   if (!area)
         return NULL;

   nr_pages = size >> PAGE_SHIFT;
   array_size = (nr_pages * sizeof(struct page *));

   area->nr_pages = nr_pages;
   area->pages = pages = kmalloc(array_size, (gfp_mask & ~__GFP_HIGH
MEM));
   if (!area->pages) {
         remove_vm_area(area->addr);
         kfree(area);
         return NULL;
   }
   memset(area->pages, 0, array_size);

   for (i = 0; i < area->nr_pages; i++) {
         area->pages[i] = alloc_page(gfp_mask);
         if (unlikely(!area->pages[i])) {
               area->nr_pages = i;
               goto fail;
         }
```

```
    }

    if (map_vm_area(area, prot, &pages))
            goto fail;
    return area->addr;

fail:
    vfree(area->addr);
    return NULL;
}
```

이 함수는 먼저 4096(페지틀크기)의 배수가 되도록 size 파라메터의 값을 반올림한다. 다음으로 vmalloc()은 get_vm_area( )를 호출하여 새 서술자를 할당하고 이 기억기령역이 할당할 선형주소를 돌려받는다. 서술자의 flags 마당를 VM_ALLOC 기발값으로 초기화한다. 이 기발는 해당 선형주소범위를 불련속적인 기억기할당을 위해 사용하려 한다는 사실을 나타낸다.(5장에서 여기서와 달리 vm_struct서술자를 하드웨어장치에 있는 기억기를 재배치(remapping)하는 용도로 사용하는것을 볼것이다.) 다음으로 vmalloc()은 alloc_pages() 를 호출하여 불련속적인 페지틀을 요청하고 불련속적인 기억기령역의 처음 선형주소를 반환하며 끝마친다.

map_vm_area() 함수는 파라메터 3개를 받는다.

area

령역을 가리키는 지적자이다.

pages

페지를 가리키는 지적자이다.

prot

할당할 페지틀의 보호비트이다. 이것을 항상 Present(존재), Accessed(접근), Read/Wfite(읽기/쓰기), Dirty(불결)에 해당하는 0x63으로 설정한다.

이 함수는 먼저 국부변수 end를 령역이 끝나는 곳의 선형주소로 설정한다.

end = address + size;

다음으로 pgd_offset_k 마크로를 사용하여 주핵심부 페지대역등록부에서 이 령역에 시작선형주소에 해당하는 입구점을 가져온다. 그후 핵심부페지표 스핀잠그기를 획득한다.

반복할 때마다 이 함수는 먼저 pmd_alloc()을 호출하여 새 령역용으로 페지중간등록부를 생성하고 이것의 물리주소를 핵심부페지대역표의 옳바른 입구점에 기록한다. 다음으로 map_area_pmd()를 호출하여 페지중간등록부와 련결되는 모든 페지표를 할당한다. address의 현재 값에 페지중간등록부 하나가 다루는 선형주소범위의 크기인 상수

2^{22}를 더하고 페지대역등록부에 대한 지적자 dir을 증가시킨다. 불련속적인 기억기령역을 참조하는 모든 페지표 입구점을 설정할 때까지 이 코드를 반복한다.

map_area_pmd()함수는 비슷한 방식으로 페지중간등록부가 가리키는 모든 페지표에 대해 반복을 한다.

pte_alloc_kernel() 함수(《페지표 다루기》 참고)는 새 페지표를 할당받아 페지중간등록부에 있는 해당 입구점을 갱신한다. 다음으로 map_area_pte()는 페지표입구점에 해당하는 모든 페지틀을 할당한다. address 값에 페지표 하나가 다루는 선형주소범위의 크기인 2^{12}를 더하고 순환을 반복한다.

map_area_pte()함수의 핵심반복코드는 다음과 같다.

```
while (address < end) {
unsigned long page;
spin_unlock(&init_mm.page_table_lock);
page_alIoc(gfp_mask);
spin_1ock(&init_mn.page_table_1ock);
if (!page)
return -ENOMEM;
set_pte(pte, mk_pte(page, prot) );
address += PAGE_SIZE;
pte++;
}
```

page_alIoc()함수를 통해 각 페지틀을 할당한다. 새로 할당한 페지틀의 물리주소를 set_pte()와 mk_pte()마크로를 리용하여 페지표에 기록한다. address에 페지틀길이인 4096을 더한 후 다시 반복한다.

vmalloc_area_pages()는 현재프로쎄스의 페지표을 건드리지 않는다는 사실에 주목하자. 그래서 어떤 프로쎄스가 핵심부방식에서 불련속적인 기억기령역에 접근하면 해당 령역에 대한 프로쎄스의 페지표입구점은 비여있으므로 페지절환이 발생한다. 그렇지만 페지절환운전기는 잘못된 선형주소를 주핵심부페지표(이것은 init_mm.pgd페지대역등록부와 이것의 자식페지표이다.)에서 검사한다. 운전기는 주핵심부페지표에서 그 주소에 대한 비여있지 않는 입구점이 있음을 발견하면 그 값을 프로쎄스의 해당 페지표입구점으로 복사하고 프로쎄스의 정상적인 실행을 재개한다. 3장에 있는 페지절환례외처리기에서 이 과정을 설명한다.

4) 불련속적인 기억기령역해제

불련속적인 기억기령역을 해제할 때에는 vfree()함수를 사용한다. 여기에 전달하는 파라메터 addr로 해제할 령역의 시작선형주소를 지정한다. vfree()는 먼저 vmlist가 가리키는 목록을 검색하여 해제할 령역과 관련한 령역서술자의 주소를 찾는다.

```
void vfree(void *addr)
{
   BUG_ON(in_interrupt());
   __vunmap(addr, 1);
}
void __vunmap(void *addr, int deallocate_pages)
{
   struct vm_struct *area;

   if (!addr)
         return;

   if ((PAGE_SIZE-1) & (unsigned long)addr) {
         printk(KERN_ERR "Trying to vfree() bad address (%p)\n",
addr);
         WARN_ON(1);
         return;
   }

   area = remove_vm_area(addr);
   if (unlikely(!area)) {
         printk(KERN_ERR "Trying to vfree() nonexistent vm area
(%p)\n",
                     addr);
         WARN_ON(1);
         return;
   }

   if (deallocate_pages) {
         int i;

         for (i = 0; i < area->nr_pages; i++) {
               if (unlikely(!area->pages[i]))
                     BUG();
               __free_page(area->pages[i]);
```

```
            }

            kfree(area->pages);
        }

        kfree(area);
        return;
    }
```

서술자의 size마당은 해제할 령역의 크기를 지정한다. 령역자체를 해제할 때에는 vmfree_area_pages()를 서술자를, 서술자를 해제할 때에는 kfree()를 호출한다.

vmfree_area_pages() 함수는 령역의 시작선형주소와 크기라는 두 파라메터를 받는다. 이 함수는 다음 코드를 실행하여 vmalloc_area_pages()가 수행한 작업을 되돌린다.

```
dir=pgd_offset_k(address);
while (address<end) {
free_area_pmd (dir, address, end-address);
address=(address+PGDIR_SIZE) & PGDIR_MASK;
dir++;
}
```

차례로 free_area_pmd()는 alloc_area_pmd()가 수행한 작업을 되돌린다.

```
while (address < end) {
free_area_pte(pmd, address, end-address);
address=(address+PMD_SIZE) & PMD_MASK;
pmd++;
}
```

다시 free_area_pte()는 alloc_area_pte()가 수행한 작업을 되돌린다.

```
while (address < end) {
pte_t page=*pte;
pte_clear(pte);
address += PAGE_SIZE;
pte++;
if (pte_none(page))
continue;
if (pte_present(page)) {
__free_page(pte_page(page));
```

```
continue;
}
printk( "Whee…Swapped out page in kernel page table\n" );
}
```

불련속적인 기억기령역에 할당한 각 페지들을 해제할 때에는 형제체계의 __free_page()함수를 리용한다. pte_clear마크로를 사용하여 페지표의 해당 입구점을 0으로 설정한다.

vmalloc()을 할 때 핵심부는 주핵심부페지대역등록부와 이것의 자식페지표의 입구점(1절에 있는 《핵심부페지표》참고)을 수정하지만 마지막 1GB를 배치하는 프로쎄스의 페지표입구점은 고치지 않는다. 이렇게 해도 핵심부는 주핵심부페지대역등록부를 뿌리로 하는 페지중간등록부와 페지표을 재사용하지 않기때문에 일없다.

례를 들어 프로쎄스가 핵심부방식에서 불련속적인 기억기령역에 접근한 후 이 령역을 해제하였다고 하자. 3장에 있는 《페지절환례외처리기》에서 설명하는 기구를 통해 프로쎄스의 페지대역등록부입구점은 주 핵심부페지대역등록부의 해당 입구점과 같아진다. 이것들은 같은 페지중간등록부와 페지표를 가리킨다. vmfree_area_pages() 함수는 페지표입구점만을 지운다.(페지표자신을 재사용하지 않는다.) 앞으로 프로쎄스가 해제한 불련속적인 기억기령역에 접근하면 페지표입구점이 비여있으므로 페지절환이 발생한다. 그런데 주핵심부페지표에 유효한 입구점이 없기때문에 운전기는 이것을 오유라고 판단한다.

제 3 절. 디스크캐쉬

이 절에서는 디스크캐쉬를 살펴본다. 즉 Linux가 디스크접근을 최대한 줄여서 체계성능을 높이려고 복잡한 기술을 어떻게 사용하는지 보여준다.

2장의 《공통파일모형》에서 살펴본것처럼 디스크캐쉬는 보통디스크에 저장한 일부 자료를 RAM에서 저장하고있도록 하는 쏘프트웨어기구로 후에 다시 이 자료에 접근할 때에는 디스크에 접근하지 않고 빨리 처리할수 있다.

Linux는 VFS가 파일경로명을 대응하는 색인마디로 변환하는 속도를 높이려고 사용하는 등록부입구점캐쉬외에 두가지 주요디스크캐쉬 즉 완충기캐쉬와 페지캐쉬를 사용한다.

이름에서도 알수 있는것처럼 《완충기캐쉬》는 완충기를 저장하는 디스크캐쉬이다. 매 완충기는 디스크블로크 하나를 저장한다. 블로크입출력연산은 디스크 접근회수를 줄이려고 완충기캐쉬를 사용한다.

한편 《페지캐쉬》는 페지를 저장하는 디스크캐쉬이다. 캐쉬의 각 페지는 정규파일이나 블로크장치파일의 여러 블로크에 대응한다. 물론 한 페지에 들어갈수 있는 블로크의 수는 블로크의 크기에 따라 다르다. 페지의 모든 블로크들은 론리적으로 련속한다.

즉 정규파일이나 블로크장치파일에서 련속된 부분이다. 핵심부는 디스크접근회수를 줄이기 위해 페지입출력연산을 시작하기 전에 요청한 자료가 이미 페지캐쉬에 저장되여있는지 검사한다. 표 4-8에서는 널리 쓰이는 입출력연산에서 완충기캐쉬와 페지캐쉬를 어떻게 사용하는지 보여준다. 일부 실례는 Ext2파일체계에 관한것이지만 대부분의 디스크기반 파일체계에 똑같이 적용한다.

표 4-8. 완충기캐쉬와 페지캐쉬 사용

핵심부함수	체계호출	캐쉬	입출력연산
bread()	없음	BufferExt2	초블로크를 읽음
bread()	없음	BufferExt2	색인마디를 읽음
generic_file_read()	getdents()	PageExt2	등록부를 읽음
generic_file_read()	read()	PageExt2	정규파일을 읽음
generic_file_write()	write()	PageExt2	정규파일에 씀
generic_file_read()	read()	Page	블로크장치파일을 읽음
generic_file_write()	write()	Page	블로크장치파일에 씀
filemap_nopage()	없음	Page	기억기사영파일접근
brw_page()	없음	Page	교환하여 내보낸 페지접근

다음에 나오는 부분들에서 이 표의 각 연산을 다룬다.

Ext2초블로크를 읽음

Ext2색인마디를 읽음

Ext2색인마디를 읽음

Ext2등록부를 읽음

Ext2정규파일을 읽음

Ext2정규파일에 쓰기

블로크장치파일을 읽음

블로크장치파일에 쓰기

기억기넘기파일접근

교환하여 내보낸 페지접근

또 이 표에서 각 류형의 입출력작업을 시작하기 위한 체계호출과 이것을 처리하는 핵심부함수를 알수 있다.

표에서 알수 있는것처럼 기억기넘기기파일과 바꿔내보내기페지를 사용하는데는 체계호출이 필요없다. 프로그람작성자는 기억기넘기기파일과 교환하여내보내기페지를 볼수

없다. 파일기억기배치를 설정하거나 교환을 활성화하면 응용프로그람은 기억기사영파일이나 교환하여내보내기페지를 마치 일반기억기처럼 사용한다. 처리기가 요청한 페지가(기억기가 아닌)디스크에 있을 때 해당 페지를 기억기로 가져올 때까지 처리기를 뒤로 연기하는 작업은 핵심부가 한다.

블로크장치파일과 정규파일을 읽기 위해 동일한 핵심부함수 generic_file_read()를 사용하고 블로크장치파일과 정규파일에 쓰기 위해 동일한 핵심부함수 generic_file_write()를 사용한다.

1. 페지캐쉬

핵심부는 불필요한 디스크접근을 피하려고 항상 페지캐쉬를 검색하여 요청한 자료가 있는지 확인하고 캐쉬에 없을 때에만 디스크에서 페지를 읽는다. 페지캐쉬를 사용하여 최대한 효률을 높이기 위해 페지캐쉬 검색은 아주 빨라야 한다.

물론 페지캐쉬에 저장되는 정보의 단위는 한 페지의 자료 전체이다. 페지가 반드시 물리적으로 린접한 디스크블로크를 소유하고있을 필요가 없으므로 페지를 장치번호와 블로크번호로 나타낼수 없다. 그 대신 페지캐쉬에 들어있는 페지를 나타내기 위해 address_space라는 자료구조의 주소와 address_space자료구조가 나타내는 파일(또는 다른 무엇)내에서의 편위를 사용한다.

1) Address_space객체

표 4-8에서 살펴본것처럼 Linux의 페지캐쉬는 여러 입출력연산의 속도를 높이기 위해 사용한다. 페지캐쉬에는 다음과 같이 여러 류형의 페지가 들어갈수 있다.

・디스크기반 파일체계의 정규파일과 등록부의 자료를 담고있는 페지.

・기억기사영파일의 자료를 담고있는 페지.

・블로크장치파일에서(파일체계계층을 건너뛰고) 직접 읽은 자료를 담고있는 페지, 핵심부는 정규파일의 자료를 담고있는 페지에 사용하는 함수와 동일한 함수를 사용해서 이것들을 처리한다.

・디스크에 교환하여내보낸 사용자방식처리기의 자료를 담고있는 페지, 이미 교환령역에 기록한 내용을 담고있는 페지캐쉬를 핵심부가 계속 유지하도록 할수 있다.

・IPC(InterProcess Communication)공유기억기 령역에 속하는 페지.

페지캐쉬의 각 페지를 어떻게 처리하겠는지 핵심부가 어떻게 알수 있는가? 례를 들어 핵심부가 페지캐쉬에 있는 페지의 내용을 수정하려 한다고 하자. 페지내용을 정규파일, 등록부, 블로크장치파일, 교환령역에서 읽는것은 서로 다른 연산이고 핵심부는 반드시 페지류형별로 적절한 연산을 실행해야 한다.

페지와 페지에 대해 동작하는 메쏘드사이의 관계를 설정하는 주요 자료구조는 address_space객체이다. 정확히 말하자면 각 address_space객체는 일반핵심부객체(소유자(owner)라고 부른다.)와 이 소유자에 속한 페지에 대해 동작하는 메쏘드사이에

련결을 설정한다.

앞에서 설명한것처럼 페지캐쉬는 다섯개 종류의 페지를 가질수 있으며 따라서 페지는 다섯개 종류의 소유자에 속할수 있다.

례를 들어 페지가 Ext2파일체계의 정규파일에 속하면 페지의 소유자는 색인마디객체이다. 이 객체의 i_maping마당은 address_space객체하나를 가리키고 이 address_space객체는 핵심부가 이 정규파일의 자료를 담고있는 페지에 대해 사용할 메쏘드집합을 정의한다.

address_space객체는 표 4-9에 렬거한 마당을 포함한다.

표 4-9. address_space객체의 마당

형	마당	설명
struct list_head	private_list	페지목록
struct list_head	i_mmap_nonlinear	비선형으로 할당된 페지목록
unsigned long	nrpages	소유자의 페지수
struct address_space _operations*	a_ops	소유자의 페지를 대상으로 하는 메쏘드
struct inode*	host	객체를 소유한 색인마디의 지시자
struct prio_tree_root	i_mmap	비공유 기억기배치의 기억기령역목록
spinlock_t	i_mmap_lock	기억기령역의 목록에 대한 스핀잠그기

private_list마당은 페지서술자목록머리부를 나타낸다. 우의 세 목록은 address_space객체의 소유자에 속한 모든 페지를 포함한다. 각 목록의 역할은 다음 절에서 설명한다. nrpages마당은 세 목록에 삽입된 모든 페지수를 나타낸다.

address_space객체의 소유자는 임의의 일반핵심부객체일수 있지만 일반적으로 VFS색인마디객체이다.(페지캐쉬는 디스크접근속도를 높이려고 도입한것이다.) 이 경우 host마당은address_space객체를 소유하는 색인마디를 가리킨다.

i_mmap, i_mmap_nonlinear, i_mmap_lock마당은 address_space객체의 소유자가 기억기사영파일의 색인마디일 때 사용된다.

address_space객체에서 가장 중요한 마당은 a_ops이다. 이 마당은 소유자의 페지를 어떻게 처리할것인지를 정의하는 메쏘드를 담고있는 address_space_operations형태의 표를 가리킨다. 메쏘드는 표 4-10과 같다.

표 4-10. address_space객체의 메쏘드

메 쏘 드	설 명
writepage	쓰기연산(페지에서 소유자의 디스크영상으로)
Readpage	읽기연산(소유자의 디스크영상에서 페지에로)
sync_page	페지에 대해 이미 순서짜기된 입출력자료전송을 시작함
prepare_write	쓰기연산을 준비함(디스크기반파일체계에서 사용)
commit_write	쓰기연산을 완료함(디스크기반파일체계에서 사용)
Bmap	파일블로크색인에서 론리적블로크번호를 얻음
flushpage	소유자의 디스크영상에서 페지를 삭제할 준비를 함
releasepage	기록형파일체계에서 페지를 해제하기 위해 사용함
direct_IO	페지의 자료를 직접 입출력 전송함

가장 중요한 메쏘드는 readpage, writepage, prepare_write, commit_write이다. 후에 이 메쏘드들을 설명한다. 대부분의 경우 메쏘드는 물리적장치에 접근하는 저수준장치구동프로그람과 소유자색인마디객체를 련결해준다. 례를 들어 정규파일의 색인마디에 대한 readpage메쏘드는 파일의 특정한 페지에 대응하는 블로크가 물리적디스크장치에 있는 위치를 어떻게 찾아내는지 알고있다.

2) 페지캐쉬자료구조

페지캐쉬는 다음의 주요자료구조를 사용한다.

페지하쉬표

address_space객체와 편위(일반적으로 파일편위)로 지정한 페지에 대해 핵심부가 페지서술자주소를 빨리 얻을수 있게 한다.

address_space객체에 있는 페지서술자목록

address_space객체가 가리키는 특정한 색인마디객체(또는 다른 핵심부객체)가 소유한 지정한 특정상태의 모든 페지를 핵심부가 빨리 얻을수 있게 한다.

페지캐쉬를 다루는 작업은 이 자료구조에 항목을 추가하고 제거하는 일, 캐쉬된 페지를 참조하는 모든 객체의 마당을 수정하는 일 등을 포함한다. pagecache_lock스핀잠그기는 다중프로쎄스체계에서 페지캐쉬자료구조를 동시에 접근하는것을 방지한다.

o 페지하쉬표

처리기가 커다란 파일을 읽을 때 페지캐쉬는 점점 이 파일의 페지로 채워지게 된다. 이런 경우 요청한 파일부분에 대한 페지를 찾으려고 페지서술자목록를 탐색하는데 오랜 시간이 걸릴수 있다.

그렇기때문에 Linux는 페지서술자지시자의 하쉬표인 page_hash_table을 사용한다. 이 하쉬표의 크기는 체계의 RAM크기에 따라 다르다. 례를 들어 128MB를 보유한 체계인 경우 page_hash_table에 페지기발 32개를 저장하고 페지서술자지시자 32768개를

포함한다.

page_hash마크로는address_space객체의 주소와 편위값에서 하쉬표에 있는 입구의 주소를 얻는다. 여기서도 충돌이 발생한 하쉬항목을 처리하기 위해 하쉬사슬(chaining)을 사용한다. 즉 페지서술자의 next_hash와 pprev_hash마당을 사용해서 하쉬값이 동일한 입구로 구성된 이중원형목록를 구성한다. page_cache_size변수는 하쉬표(결국 전체 페지캐쉬)의 충돌목록에 포함된 페지서술자의 수를 나타낸다.

add_page_to_hash_queue()와 remove_page_from_hash_queue()함수는 항목을 하쉬표에 추가하고 표에서 항목을 삭제한다.

o **address_space객체에 있는 페지서술자목록**

이미 살펴본것처럼 address_space객체는 3가지 페지서술자목록를 포함한다. clean_pages, dirty_pages, locked_pages마당에 각 목록의 머리부가 들어있다. 핵심부는 이 목록를 사용하여 특정상태에 있는 파일의 모든 페지를 빨리 찾을수 있다.

clean_pages

잠그기가 걸려있지 않으며 불결하지 않은 페지를 포함한다.(페지서술자의 PG_locked와 PG_dirty기발은 0이다.) PG_uptodate기발은 페지에 있는 자료가 최신(up_to_date)인 페지를 나타낸다. 디스크영상에서 아직 페지내용을 읽지 않았다면 페지는 최신상태가 아니다.

dirty_pages

최신자료를 담고있는 페지를 포함한다. 그러나 디스크의 영상은 수정되지 않았다. 페지서술자의 PG_uptodate와 PG_dirty기발은 설정되고 PG_locked기발은 설정되지 않는다.

locked_pages

디스크에 읽거나 쓰는중인 내용을 담고있는 페지를 포함한다. 따라서 페지에 접근할 수 없다. PG_locked 기발이 설정된다.

add_page_to_inode_queue()함수는 페지서술자를 address_space객체의 clean_pages목록에 삽입한다. remove_page_from_inode_queue()는 페지서술자를 현재 담고있는 목록에서 삭제한다. 핵심부는 페지의 상태가 바뀔 때마다 페지서술자를 한 목록에서 다른 목록으로 옮긴다.

o **페지캐쉬관련 페지서술자마당**

페지가 페지캐쉬에 들어있을 때 페지서술자의 일부마당은 특수한 의미를 나타낸다.

list

페지상태에 따라 각각 address_space객체의 clean, dirty, locked 상태인 페지를 담고있는 2중련결목록의 이전, 다음 요소를 나타내는 지시자를 포함한다.

mapping

페지가 속한 address_space객체를 가리킨다. 페지가 페지캐쉬에 속하지 않으면 이

마당은 NULL이다.

index

페지소유자가 색인마디객체면 디스크영상에서 페지에 들어있는 자료의 위치를 나타낸다. 이 값의 단위는 페지의 크기이다.

이 함수의 이름은 이전 판본인 2.2핵심부에서 유래되였다.

next_hash

페지하쉬목록에서 충돌하는(하쉬값이 같은) 다음페지서술자를 가리킨다.

pprev_hash

페지하쉬목록에서 충돌하는 이전페지서술자의next_hash마당을 가리킨다. 또한 페지캐쉬에 페지를 삽입할 때 대응하는 페지서술자의 사용계수기(count마당)를 증가시킨다. count마당값이 정확히 1이면 페지는 캐쉬에 속하지만 어떤 처리기도 이 페지에 접근하고있지 않다. 따라서 여유기억기가 모자라면 언제든지 이 페지를 페지캐쉬에서 삭제할수 있다.

3) 페지캐쉬처리함수

페지캐쉬를 사용하는 고수준함수들은 페지를 탐색하고 추가하고 제거한다.

find_get_page마크로는 address_space객체의 주소와 편위값을 변수로 받는다. 이 마크로는 page_hash마크로를 사용하여 변수의 값에 대응하는 하쉬표입구의 주소를 얻고 __find_get_page()함수를 호출하여 적절한 충돌목록에서 요청한 페지서술자를 찾는다. __find_get_page()는 pagecache_lock스핀잠그기를 얻고 하쉬값이 같은 입구의 목록를 탐색한 다음 스핀잠그기를 해제한다. 페지를 찾으면 대응하는 페지서술자의 count마당을 증가시키고 그 주소를 반환한다. 그렇지 않으면 NULL을 반환한다.

add_to_page_cache()함수는 새로운 페지서술자(주소는 변수로 전달한다.)를 페지캐쉬에 삽입한다. 이 작업은 다음과 같은 과정을 거친다.

1. pagecache_lock스핀잠그기를 얻는다.

2. 페지기발의 PG_uptodate, PG_error, PG_dirty, PG_referenced, PG_arch_1, PG_checked기발을 지우고 페지기발의 PG_locked기발을 설정한다. 즉 페지가 잠그기되여있으며 캐쉬에 들어있고 아직 자료로 채워지지 않았음을 나타낸다.

3. 페지서술자의 count마당을 증가시킨다.

4. 페지서술자의 index마당을 변수로 넘겨준 값으로 초기화한다. 이 값은 페지디스크영상에서 페지에 들어있는 자료의 위치를 나타낸다.

5. add_page_to_inode_queue()를 호출하여 페지서술자를 address_space객체의 clean_pages목록에 삽입한다. Address_space객체의 주소는 변수로 전달된다.

6. add_page_to_hash_queue()를 호출하여 페지서술자를 하쉬표에 삽입한다. address_space 객체의 주소와 페지의 index마당의 값을 하쉬열쇠로 사용한다.

7. pagecache_lock스핀잠그기를 해제한다.

8. lru_cache_add()를 호출하여 페지서술자를 비활성(inactive)목록에 추가한다.

find_or_create_page()함수는 find_get_page와 류사하다. 그렇지만 요청한 페지가 캐쉬에 없을 때 alloc_page()를 호출하여 새로운 페지기발을 얻은 다음 add_to_page_cache()를 호출하여 페지서술자를 페지캐쉬에 삽입한다.

remove_inode_page()함수는 페지서술자를 페지캐쉬에서 제거한다. 이 함수는pagecache_lock 스핀잠그기를 얻은 다음 remove_page_from_inode_queue(), remove_page_from_hash_queue()를 호출하고 스핀잠그기를 해제한다.

2. 완충기캐쉬

완충기캐쉬(Buffer cache)의 핵심은 상대적으로 느린 디스크가 자료를 읽고 쓰는 작업이 끝날 때까지 처리기가 기다리지 않고 다른 일을 수행할수 있도록 처리기를 놓아주는것이다. 따라서 한번에 많은 자료를 쓰는것이 오히려 비생산적일수도 있다. 대신 자료를 조금씩 일정한 간격으로 기록하여 입출력연산이 사용자처리기의 성능과 사용자가 느끼는 응답시간에 주는 영향을 최소한으로 줄여야 한다.

핵심부는 완충기 기록속도를 조절하려고 매 완충기에 관한 많은 정보를 관리한다. 여기에는 기억기내에 있는 완충기가 바뀌였으므로 기록해야 함을 나타내는 《불결한》비트와 디스크에 흘리기 전에 기억기에 완충기를 얼마동안이나 저장해놓을지를 나타내는 시간형를 포함한다. 완충기에 대한 정보는 완충기머리부에 저장한다. 이 자료구조는 사용자자료완충기와 함께 잘 관리해야 한다.

완충기캐쉬의 크기는 가변적이다. 사용할수 있는 완충기가 없을 때 새로운 완충기에 대한 요청이 들어오면 페지틀을 할당한다. 반대로 여유기억기가 모자랄 때 완충기를 해제하고 해당 페지틀을 재사용한다.

완충기캐쉬는 두가지 자료구조로 구성된다.

・캐쉬에 들어있는 완충기를 나타내는 완충기머리부 집합

・주어진 장치와 블로크번호쌍에 대응하는 완충기를 나타내는 완충기머리부를 핵심부가 빨리 찾도록 도와주는 하쉬표

1) 완충기머리부자료구조

매 완충기머리부의 자료구조는 buffer_head이다. 이 자료구조에는 자신만의 스랩할당자캐쉬인 bh_cachep를 포함한다. 이 캐쉬를 완충기캐쉬자체와 혼돈하면 안된다. 스랩할당자캐쉬는 완충기머리부객체를 위한 기억기캐쉬이며 이 캐쉬는 디스크와 호상작용하지 않으며 기억기를 효률적으로 관리하기 위한 기법에 불과하다. 반대로 완충기캐쉬는 완충기에 들어있는 자료를 위한 디스크캐쉬이다.

블로크장치장치구동프로그람이 사용하는 각 완충기마다 완충기의 현재 상태를 나타내는 완충기머리부가 반드시 있어야 한다. 완충기머리부는 《사용하지 않음》상태인 경우가 있으며 이 경우 대응하는 완충기가 없다. 핵심부는 어느 정도 《사용하지 않음》

상태인 완충기머리부를 가지고있어서 지속적으로 기억기를 할당하고 해제하는 상태를 피한다.

일반적으로 완충기머리부는 다음 상태중 하나이다.

○ 사용하지 않는 완충기머리부

이 객체를 사용할수 있다. 마당값은 무의미하다. b_dev마당의 값은 B_FREE(0xffff)이다.

○ 캐쉬된 완충기에 대한 완충기머리부

b_data마당은 완충기캐쉬에 저장된 완충기를 가리킨다. b_dev마당은 블로크장치를 나타낸다. BH_Mapping기발이 설정된다. 그리고 완충기는 다음 상태중 하나일수 있다.

최신상태가 아니다.(BH_Uptodate기발이 지우기 상태)

완충기의 자료가 유효하지 않다.(례를 들면 자료를 디스크에서 아직 읽지 않은 경우)

불결하다.(BH_Dirty기발이 설정됨)

완충기의 자료가 변경되였으며 디스크의 해당 블로크를 갱신해야 한다.

잠그기상태다.(BH_Lock기발이 설정됨)

완충기에 대해 입출력자료전송이 진행중이다.

2) 비동기완충기머리부

비동기완충기머리부의 b_data마당은 페지입출력연산을 위해 사용하는 페지의 완충기를 가리킨다. 이 경우 BH_Async기발이 설정된다. 페지입출력연산을 완료하면 BH_Async기발이 삭제되지만 완충기머리부는 해제하지 않는다. 대신에 할당된채로 페지의 단순련결원형목록에 삽입된다. 따라서 매번 새로 할당하는 부하없이 재사용할수 있다. 비동기완충기머리부는 핵심부가 기억기를 회수할 때 언제나 해제된다. 엄밀히 말하면 완충기캐쉬 자료구조는 캐쉬완충기에 대한 완충기머리부를 가리키는 지시자만 포함한다. 완충기캐쉬에서 볼수 있는 완충기머리부뿐만아니라 모든 종류의 완충기머리부를 다루기 위해 핵심부에서 사용하는 자료구조와 메쏘드를 고찰한다.

○ 사용하지 않는 완충기머리부목록

사용하지 않는(unused) 완충기머리부는 단순련결목록에 저장한다. unused_list변수가 목록의 첫번째 항목을 가리킨다. 목록내에 있는 다음입구의 주소는 각 완충기머리부의 b_next_free마당에 저장한다. 목록의 현재입구수는 nr_unused_buffer_heads변수에 저장한다. 다중프로쎄스체계에서는 동시접근을 방지하려고 unused_list_lock 스핀잠그기를 사용한다.

사용하지 않는 완충기머리부의 목록은 완충기머리부객체를 위한 1차적인 기억기캐쉬로 동작하며 bh_cachep스랩할당자 캐쉬는 2차적인 기억기캐쉬로 동작한다. 완충기머리부가 더는 필요하지 않으면 사용하지 않는 완충기머리부목록에 삽입한다. 완충기머리부는 목록입구수가 MAX_UNUSED_BUFFERS(보통 100입구)를 넘는 경우에만 해제되여 스랩할당자로 들어간다.(핵심부가 완충기머리부와 관련한 기억기를 해제하는 단계

이다.) 다시말해서 이 목록에 들어있는 완충기머리부는 스랩할당자립장에서는 할당된 객체로 간주하며 완충기캐쉬립장에서는 사용하지 않는 자료구조로 간주한다.

목록의 입구중 NR_RESERVED(보통 80)개는 페지입출력연산을 위해 예약되여있다. 리유는 여유완충기머리부가 없어서 발생할수 있는 위험한 교착(deadlock)을 방지하기 위해서이다. 핵심부는 여유기억기가 모자라면 페지를 디스크로 바꿔내보내기 하여 페지들을 해제하려 한다. 이 과정에 페지입출력 파일연산을 위해 적어도 추가완충기머리부 하나가 필요하다. 교환알고리듬이 완충기머리부를 얻지 못하면 계속 기다리면서 완충기를 해제할수 있도록 파일에 대한 쓰기를 지속해야 한다. 진행중인 파일연산을 완료하면 완충기머리부가 적어도 NR_RESERVED개 해제되기때문이다.

새로운 완충기머리부를 얻기 위해 get_unused_buffer_head()함수를 호출한다. 이 함수는 다음과 같은 연산을 수행한다.

1. unused_list_lock스핀잠그기를 얻는다.

2. 사용하지 않는 완충기머리부목록이 항목을 NR_RESERVED개이상 포함하고있으면 그중 하나를 목록에서 제거하고 스핀잠그기를 해제한 다음 머리부의 주소를 반환한다.

3. 그렇지 않으면 스핀잠그기를 해제하고 우선순위 GFP_NOFS로 kmem_cache_alloc()을 호출하여 bh_cachep스랩할당자캐쉬에서 새로운 완충기머리부를 할당한다. 이 연산이 성공하면 그 주소를 반환한다.

4. 여유기억기가 없다. 완충기입출력연산을 위해 완충기머리부를 요청한 경우라면 NULL(실패)을 반환한다.

5. 여기까지 왔다면 페지입출력연산을 위해 완충기머리부를 요청한 경우이다. 사용하지 않는 완충기머리부의 목록이 비여있지 않다면 unused_list_lock스핀잠그기를 얻고 입구 하나를 제거하고 스핀잠그기를 해제하고 완충기머리부의 주소를 반환한다.

6. 그렇지 않고 목록이 비여있다면 NULL(실패)을 반환한다.

put_unused_buffer_head()함수는 완충기머리부를 해제하는 반대연산을 실행한다. 이 함수는 사용하지 않는 완충기머리부목록이 MAX_UNUSED_BUFFERS보다 적은 항목을 포함하고있다면 객체를 목록에 삽입한다. 그렇지 않으면 완충기머리부에 대해 kmem_cache_free()를 호출하여 객체를 해제하여 스랩할당자에 넣는다.

○ 캐쉬완충기에 대한 완충기머리부목록

완충기가 완충기캐쉬에 속한 경우 완충기머리부의 기발이 완충기의 현재상태를 알아낸다. 례를 들어 블로크가 캐쉬에 없으면 디스크에서 블로크를 읽어야 하는데 새로운 완충기를 할당하고 완충기의 내용이 의미가 없으므로 완충기머리부의 BH_Uptodate기발을 지운다. 디스크에서 읽어서 완충기를 채우는 동안 BH_Lock기발은 1로 설정되여 완충기가 해제되는것을 막는다. 읽기연산을 성공적으로 마치면 BH_Uptodate기발은 1로 설정되고 BH_Lock기발을 지운다. 디스크에 블로크를 기록해야 하는 경우 완충기내용이 바뀌므로 BH_Dirty기발을 1로 설정한다. 기발은 완충기가 성공적으로 디스크에 기

록된 다음에만 지워진다.

사용중인 완충기와 관련한 모든 완충기머리부는 2중련결목록에 저장한다. b_next_free와 b_prev_free 마당이 2중련결목록을 구성한다. 3가지 다른 목록이 있는데 마크로 정의한 색인으로 식별한다.(BUF_CLEAN, BUF_DIRTY, BUF_LOCKED)이 목록들을 곧 정의한다.

불결한 완충기를 디스크로 흘리는 속도를 높이려고 세 목록를 도입하였다.(뒤부분에 있는 《디스크에 불결한 완충기 기록하기)를 참고) 효률성을 높이기 위해 완충기머리부는 상태가 바뀔 때 한 목록에서 다른 목록으로 즉시 이동하지 않는다.

BUF_CLEAN

이 목록에는 불결하지 않는 완충기(BH_Dirty기발이 off)의 완충기머리부를 포함한다. 반드시 최신목록완충기일 필요는 없다. 즉 반드시 유효한 자료를 포함해야 할 필요가 없다는 의미이다. 최신완충기가 아니라면 잠그기걸린 상태(BH_Lock on)거나 이 목록에 있는동안 물리적장치로부터 읽도록 선택했을수도 있다. 이 목록에 있는 완충기머리부는 반드시 불결한것이 아니다. 즉 불결한 완충기를 디스크에 흘리는 함수는 대응하는 완충기를 무시한다.

BUF_DIRTY

이 목록에는 주로 물리적장치에 기록하도록 선택하지 않은 불결한 완충기의 완충기머리부를 포함한다. 즉 아직 블로크장치구동프로그람에 대한 블로크요청에 포함되지 않은 불결한 완충기를 포함한다.(BH_Dirty는 on이고 BH_Lock은 off이다) 그러나 이 목록은 불결하지 않는 완충기도 포함하는데 어떤 경우 디스크에 흘리지도, 완충기머리부를 목록에서 제거하지도 않은 상태에서 불결한 완충기의 BH_Dirty기발이 지워지는 경우가 있기때문이다.(례를 들면 플로피디스크를 탑재해제 하지 않고 구동프로그람에서 제거했을 때이다. 물론 이 경우 대부분은 자료를 잃게 된다.)

BUF_LOCKED

이 목록에는 주로 블로크장치에서 읽거나 블로크장치에 쓰기 위해 선택한 불결한 완충기의 완충기머리부를 포함한다.(BH_Lock은 on이다. add_request() 함수가 블로크요청에 완충기머리부를 포함하기 전에 이 값을 초기화하므로 BH_Dirty는 지워진다.) 그러나 어떤 잠그기가 걸린 완충기에 대한 쓰기연산을 완료하면 저수준블로크장치운전기는 완충기머리부를 목록에서 제거하지 않고 BH_Lock 기발을 지운다. 이 목록의 완충기머리부는 불결하지 않거나 불결하지만 기록을 위해 선택한 상태이다.

사용중인 완충기와 관련한 완충기머리부의 경우 해당 완충기머리부의 b_list마당은 완충기를 포함하는 목록의 색인을 지정한다. lru_list배렬은 각 목록의 첫번째 입구주소를 저장하고 ar_buffers_type배렬은 각 목록의 입구수를 저장한다. size_buffers_type 배렬은 (바이트 단위로)각 목록의 전체 용량을 저장한다. lru_list_lock스핀잠그기는 다중프로쎄스체계에서 배렬에 동시에 접근하는것을 막는다.

mark_buffer_dirty()와 mark_buffer_clean()함수는 각각 완충기머리부의 BH_Dirty기발을 설정하거나 지운다. 체계전체의 불결한 완충기의 수를 제한하기 위해 mark_buffer_dirty()는 balance_dirty()함수를 호출한다.(《불결한 완충기를 디스크에 기록하기》참고) 두 함수 모두 refill_buffer()를 호출한다. 이 함수는 BH_Dirty와 BH_Lock기발의 값에 따라 완충기머리부를 적절한 목록으로 옮긴다.

핵심부는 BUF_DIRTY목록 외에도 각 색인마디객체마다 불결한 완충기의 2중련결목록 두개를 관리한다. 이것들은 핵심부가 해당 파일의 모든 불결한 완충기를 흘려야 할 때 사용한다. 례를 들면 fsync()나 fdatasync()봉사호출을 처리할 때다.(뒤에 나오는 《sync(), fsync(), fdatasync()체계호출》참고)

두 목록중에서 한 목록은 파일의 조종자료(디스크 색인마디와 같은)를 담고있는 완충기들을 포함하고 다른 목록은 파일의 자료를 담고있는 완충기들을 포함한다. 이 목록들의 머리부는 각각 색인마디객체의 마당에 저장된다. 완충기머리부의 b_inode_buffers마당은 목록의 다음과 이전 요소를 가리킨다. 두 목록은 lru_list_lock스핀잠그기로 보호한다. buffer_insert_inode_queue()와 buffer_insert_inode_data_queue()함수를 사용하여 완충기머리부를 i_dirty_buffers와 i_dirty_data_buffers_lists에 삽입한다. inode_remove_queue()함수는 완충기머리부를 포함하는 목록에서 완충기머리부를 제거한다.

○ **캐쉬된 완충기머리부의 하쉬표**

완충기캐쉬에 속한 완충기머리부주소는 큰 하쉬표에 삽입된다. 핵심부는 장치식별자와 블로크번호로부터 대응하는 완충기머리부의 주소를 얻기 위해 하쉬표을 사용한다.

완충기머리부를 자주 검사하므로 하쉬표가 핵심부의 속도를 상당히 향상한다. 완충기입출력연산을 시작하기 전에 핵심부는 반드시 요청한 블로크가 이미 완충기캐쉬에 있는지 검사해야 한다. 이때 하쉬표는 핵심부가 캐쉬된 완충기목록를 오래동안 순차적으로 탐색하는 상황을 막아준다.

하쉬표는 hash_table배렬에 들어있고 체계초기화과정에 할당된다. 한편 그 크기는 체계에 설치된 기억기의 량에 따라 달라진다. 례를 들어 128MB RAM을 보유한 체계에서 hash_table은 페지기발 4개에 저장되며 완충기머리부지시자 4096개를 포함한다. 충돌한 입구들은 각 완충기머리부의 b_next와 b_pprev 마당에 의해 2중련결목록으로 련결된다. hash_table_1ock 읽기/쓰기 스핀잠그기는 다중프로쎄스체계에서 하쉬표자료구조를 동시에 접근하는것을 막는다.

get_hash_table()함수는 하쉬표에서 완충기머리부를 찾는다. 찾으려는 완충기머리부를 장치번호, 블로크번호, 대응하는 자료블로크의 크기라는 세 변수로 나타낸다. 함수는 장치번호와 블로크번호값을 해싱하여 하쉬표에서 충돌목록의 첫번째 항목을 찾는다. 그리고 목록에 있는 각 입구의 b_dev, b_blocknr, b_size 마당을 검사하여 요청한 완충기머리부의 주소를 반환한다. 완충기머리부가 캐쉬에 없으면 함수는 NULL을 반환한다.

3) 완충기사용계수기

완충기머리부의 b_count 마당은 대응하는 완충기의 사용계수기이다. 계수기는 완충기에 대한 연산을 수행하기 직전에 증가하고 수행한 직후에 감소한다. 계수기는 주로 보호를 위한 잠그기로 사용하는데 핵심부는 완충기의 사용계수기가 0이 아닌동안에는 완충기(또는 그 내용)를 제거하지 않기때문이다. 대신 주기적으로 또는 여유기억기가 모자라면 캐쉬된 완충기를 검사하여 계수기가 0인 완충기를 제거한다. 바꾸어 말하면 사용계수기가 0인 완충기는 완충기캐쉬에 속할수 있지만 해당 완충기가 얼마나 캐쉬에 머물러 있을지는 알수 없다.

핵심부조종경로가 완충기에 접근하려면 먼저 사용계수기를 증가시켜야 한다. 이 작업은 일반적으로 완충기를 찾는 getblk()함수를 호출하여 처리하므로 계수기를 증가시키는 작업을 고수준함수가 명시적으로 처리할 필요는 없다. 핵심부조종경로가 완충기에 접근하는것을 멈추면 brelse() 또는 blorget()을 호출하여 대응하는 사용계수기를 감소시킨다. 두 함수의 차이점은 bforget()은 완충기를 깨끗한 상태로 표시하여 아직 디스크에 기록하지 않은 변경된 완충기내용을 핵심부가 처리하도록 한다.

4) 완충기페지

페지캐쉬와 완충기캐쉬가 서로 다른 디스크캐쉬이지만 Linux 2.6 판본에서는 서로 공유되여있다.

사실 효률성때문에 각 완충기의 할당 단위는 개별 기억기객체단위가 아니다. 대신에 완충기는 《완충기페지》라는 전용페지에 저장된다. 한 완충기페지안에 있는 모든 완충기는 같은 크기여야 한다. 따라서 80x86구조에서 완충기페지는 블로크크기에 따라 완충기를 1개부터 8개까지 포함할수 있다(1, 2, 4, 8개).

더 중요한 제약조건은 한 완충기페지의 모든 완충기는 반드시 블로크장치에서 린접한 블로크에 해당해야 한다는 점이다. 례를 들어 핵심부가 어떤 정규파일의 1kB색인마디 블로크를 읽으려고 한다고 가정하자. 핵심부는 색인마디를 저장하기 위해 1kB완충기를 하나 할당하는것이 아닌 완충기 4개를 저장할수 있는 한 페지전체를 예약해야 한다. 이 완충기는 요청한 색인마디블로크를 포함해서 블로크장치의 린접한 4개 블로크그룹의 자료를 포함하게 된다.

두가지 다른 방법으로 완충기캐쉬를 취급한다는것은 리해하기 쉽다. 한가지는 완충기를 저장하는 저장소(container)역할로 이때에는 완충기캐쉬를 통해 완충기에 개별적으로 접근할수 있다. 다른 한가지는 각 완충기페지는 블로크장치파일의 4kB부분을 가질수 있으며 따라서 페지캐쉬에 포함된다. 바꾸어 말하면 완충기캐쉬에 캐쉬된 RAM령역은 언제나 페지캐쉬에 캐쉬된 RAM령역의 부분집합이다. 이 기구는 완충기캐쉬와 페지캐쉬 사이의 동기화문제를 획기적으로 줄여주는 우점이 있다.

2.2 판본핵심부에서는 두가지 디스크 캐쉬가 서로 공유되여있지 않았다. 어떤 물리적블로크는 RAM에서 하나는 페지캐쉬에 그리고 또 하나는 완충기캐쉬에 영상 두개를

가질수 있었다. 자료손실을 방지하려고 두 블로크의 기억기영상중 하나가 변경되면 2.2 핵심부는 반드시 다른 하나를 찾아서 변경해야 하였다. 짐작할수 있지만 이것은 비용이 많이 드는 작업이다.

반면에 Linux 2.6에서 완충기를 변경하는것은 완충기를 포함하는 페지를 변경하는 것이고 반대경우도 마찬가지이다. 핵심부는 단지 완충기머리부와 페지서술자의 불결한 기발만 잘 살피면 된다. 례를 들어 완충기머리부가 불결한것으로 표시되면 핵심부는 반드시 해당 완충기를 포함하는 페지의 PG_dirty 기발을 설정해야 한다.

완충기머리부와 페지서술자는 완충기페지와 대응하는 완충기사이의 련결을 정의하는 마당 몇개를 포함한다. 페지를 완충기페지로 사용할 때 페지서술자의 buffers마당은 페지에 포함된 첫번째 완충기의 완충기머리부를 가리킨다. 그렇지 않으면 buffers는 NULL이다. 또 각 완충기머리부의 b_this_page마당은 완충기페지에 포함된 모든 완충기의 완충기머리부를 포함하는 단순련결목록를 구성한다. 그림 4-20은 완충기 4개를 담고있는 완충기페지와 각 완충기에 대응하는 완충기머리부를 보여준다.

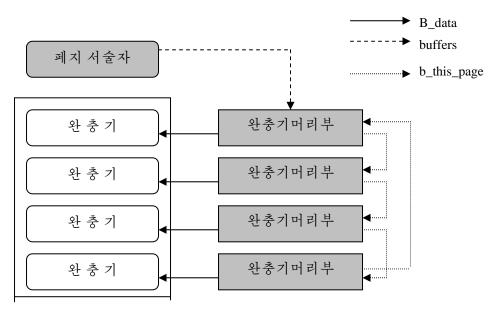



그림 4-20. 완충기 4 개를 담고있는 완충기페지와 각 완충기의 완충기머리부

다음과 같은 례외적인 경우가 있다. 페지입출력연산과 관련한 페지의 경우(《페지입출력연산 》 참고) 핵심부가 비동기완충기머리부 몇개를 할당하고 buffers와 b_this_page마당을 사용하여 이것들을 페지에 련결할수 있다. 따라서 어떤 페지에 대응하는 완충기하드가 완충기캐쉬에 들어있지 않으면서도 이 페지를 완충기페지로 사용할수 있다.

○ **완충기페지할당**

핵심부는 완충기캐쉬가 해당 블로크에 대한 자료를 담고있지 않은 경우 새로운 완충기페지를 할당한다. 핵심부는 이를 위해 grow_buffers()함수를 호출한다. 이 함수는 블로크를 식별하기 위한 변수 3개를 받는다.

· 블로크장치번호: 장치의 주번호, 부번호

· 론리적블로크번호:블로크장치안에서 블로크의 위치

· 블로크크기

함수는 다음과 같은 작업을 수행한다.

1. 요청한 블로크를 포함하는 블로크장치안에서 자료의 페지의 편위 index를 계산한다.

2. 블로크장치서술자의 주소 bdev를 얻는다.(《블로크장치구동프로그람관리》 참고)

3. 필요하다면 grow_dev_page()를 호출하여 새로운 완충기페지를 생성한다. 이 함수는 다음단계를 수행한다.

a. 블로크장치의 address_space 객체(bdev->bd_inode->i_mapping)와 페지편위색인을 변수로 find_or_create_page()를 호출한다. 앞에서 본 《페지캐쉬처리함수》에서 설명한것처럼 find_or_create_page()는 페지캐쉬에서 페지를 탐색하고 필요하다면 새로운 페지를 캐쉬에 삽입한다.

b. 이제 해당 페지의 서술자가 정확히 페지캐쉬에 들어있다. 함수는 서술자의 buffers 마당을 검사하여 NULL이면 페지에 아직 완충기가 들어있지 않은 경우므로 3e 단계로 이동한다.

c. 페지에 들어있는 완충기의 크기가 요청한 블로크의 크기와 같은지 검사한다. 같다면 페지서술자의 주소를 반환한다.(페지캐쉬에서 찾은 페지가 유효한 완충기페지이다.)

d. 그렇지 않으면 try_to_free_buffers()를 호출하여 페지에 있는 완충기들을 해제할수 있는지 검사한다. 함수가 실패하면 다른 처리기가 완충기를 시용하고있는것이므로 grow_dev_page()함수는 NULL을 반환한다.(요청한 블로크에 대해 완충기페지를 할당할수 없다.)

e. create_buffers()함수를 호출하여 페지안에 요청한 크기의 블로크를 위한 완충기머리부를 할당한다. 페지의 첫번째 완충기에 대한 완충기머리부의 주소는 페지서술자의 buffers 마당에 저장하고 모든 완충기머리부는 완충기머리부의 b_this_page마당으로 실현한 단순련결원형목록에 삽입한다. 또 완충기머리부의 b_page 마당을 페지서술자의 주소로 초기화한다.

f. 페지서술자의 주소를 반환한다.

4. grow_dev_page()가 NULL을 반환했으면 0을 반환한다.(실패)

5. hash_page_buffers()함수를 호출하여 완충기페지의 단순련결원형목록의 모든 완충기머리부의 마당을 초기화하고 이것들을 완충기캐쉬에 삽입한다.

6. 페지의 잠그기를 풀어준다.(find_or_create_page()에서 페지에 잠그기를 건다.)

7. 페지의 사용계수기를 감소시킨다(역시 find_or_create_page()에서 계수기를 증가시킨다).

8. 완충기페지의 전체수를 저장하고있는 buffermem_pages변수를 증가시킨다. 이 값은 페지크기를 단위로 했을 때 완충기캐쉬에 캐쉬된 기억기의 크기이다.

9. 1을 반환한다(성공).

◦ getblk()함수

getblk()함수는 완충기캐쉬에 대한 주봉사루틴이다. 핵심부가 물리적장치의 블로크내용을 읽거나 쓰기 전에 반드시 요청한 완충기에 대한 완충기머리부가 이미 완충기캐쉬에 들어있는지 검사해야 한다. 완충기가 캐쉬에 없다면 핵심부는 반드시 새로운 항목을 캐쉬에 생성해야 한다. 이를 위해 핵심부는 장치식별자, 블로크번호, 블로크크기를 변수로 getblk()를 호출한다. 이 함수는 완충기에 대응하는 완충기머리부의 주소를 반환한다.

완충기머리부가 캐쉬에 있다는 사실이 완충기의 자료가 유효하다는것을 의미하지는 않는다.(례를 들면 아직 디스크에서 완충기를 읽지 않았을수도 있다.) 블로크를 읽는 모든 함수는 getblk()을 통해서 얻은 완충기가 최신인지 검사해야 한다. 최신이 아니면 완충기를 사용하기 전에 디스크에서 블로크를 읽어야 한다.

getblk()함수는 다음과 같이 간단하다.

```
sturct buffer_head* getblk(kdev_t dev, int block, int size)
{
for(;;)
{
struct buffer_head* bh;
bh=get_hash_table(dev, block, size);
if(bh)
return bh;
if (!grow_buffers(dev, block, size))
free_more_memory();
}
}
```

이 함수는 먼저 get_hash_table()을 호출하여(《캐쉬된 완충기머리부의 하쉬표》참고) 요청한 완충기머리부가 이미 캐쉬에 있는지 검사한다. 캐쉬에 있다면 발견한 완충기머리부주소를 반환한다.

요청한 완충기머리부가 캐쉬에 없으면 getblk()는 grow_buffefs()를 호출하여 요청한 블로크에 대한 완충기를 포함하는 새로운 완충기페지의 할당을 시도한다. grow_buffers()

가 페지할당을 실패하면 getblk()는 기억기회수를 시도한다. 그리고 get_hash_table()이 요청한 완충기를 완충기캐쉬에서 찾을 때까지 이 작업을 계속 반복한다.

○ 디스크에 불결한 완충기기록하기

Unix체계는 불결한 완충기를 블로크 장치에 기록할 때 《쓰기지연(deferred writing)》을 허용한다. 쓰기지연은 체계성능을 매우 향상시킨다. 완충기에 대한 쓰기연산 여러번을 해당 디스크블로크에 대해 한번 느리게 물리적으로 갱신하여 만족시킬수 있다. 더구나 프로쎄스가 일반적으로 쓰기지연때문에 보류되지 않으며 주로 읽기지연때문에 보류되므로 쓰기연산은 읽기연산만큼 중요하지 않다. 쓰기지연으로 하여 물리적블로크장치는 일반적으로 쓰기요청보다 훨씬 많은 읽기요청을 처리한다.

불결한 완충기는 기능한 마지막순간 즉 체계끄기시간까지 주기억기에 머물수도 있다. 그렇지만 쓰기지연을 이렇게 극단적으로 추구하면 두가지 주요부족점이 있다.

・하드웨어 또는 전원에 문제가 발생하면 RAM의 내용을 더는 읽을수 없으며 체계를 시작한 시점부터 수행한 많은 파일변경내용을 잃어버리게 된다.

・완충기캐쉬의 크기 즉 완충기캐쉬를 저장할 RAM이 매우 커야 한다. 최소한 접근하는 블로크장치크기정도는 되여야 한다.

따라서 다음과 같은 경우에 불결한 완충기를 디스크에 흘리기(기록)한다.

・완충기캐쉬가 꽉 찼고 새로운 완충기가 필요할 때 또는 불결한 완충기의 수가 너무 커졌을 때. 둘중 한가지 경우가 발생하면 bdflush 핵심부스레드가 동작한다.

・완충기가 불결한 상태로 너무 오래 있는 경우. kupdate핵심부스레드가 주기적으로 오래된 완충기를 흘린다.

・블로크장치의 모론 완충기 또는 특정파일의 모든 완충기를 흘리기하도록 처리기가 요청하는 경우. 처리기는 sync(), fsync(), fdatasync() 체계호출을 실행하여 흘리기를 요청할수 있다.

《완충기캐쉬》에서 설명한것처럼 완충기페지의 일부 완충기가 불결한면 완충기페지는 불결하다.(PG_DIRTY 기발이 설정된다.) 핵심부가 완충기페지의 모든 불결한 완충기를 디스크에 흘리고 나면 즉시 페지의 PG_DIRTY기발을 초기화한다.

○ bdflush핵심부스레드

bdflush핵심부스레드(kflushd라고도 한다)는 체계초기화과정에서 생성된다. 이 스레드는 bdflush() 함수를 호출해서 불결한 완충기의 일부를 선택하여 완충기에 대응하는 블로크의 변경내용을 물리적 블로크장치에 기록하도록 한다.

bdflush 동작은 몇개의 체계파라메터로 조종한다. 이것들은 bdf_prm표의 b_un 마당에 저장되며 /proc/sys/vm/bdflush 파일이나 bdflush() 체계호출을 통해 접근할수 있다. 각 파라메터는 기본값이 있으며 bdflush_min과 bdflush_max표에 저장된 최소, 최대값사이에서 변경할수 있다. 파라메터는 표 4-11과 같다.

교육성 교육정보쎈터

표 4-11. 완충기캐쉬튜닝(tunning)파라메터

파라메터	기본값	최소값	최대값	설 명
nfract	40	0	100	bdflush를 깨우게 하는 불결한 완충기의 비률
nfract_sync	60	0	100	bdflush를 차단(동기적으로 실행)방식으로 깨우게 하는 불결한 완충기의 비률
age_buffer	3000	100	600000	불결한 완충기를 디스크에 기록할 때까지의 시간넘침시간(틱크단위)
interval	500	0	1000000	kupdata가 활성화하는 시간간격(틱크단위)

틱크단위는 보통 10ms이다.

핵심부스레드는 다음과 같은 몇가지 특별한 경우에 깨여난다.

・balance_dirty() 함수에서 BUF_DIRTY와 BUF_LOCKED 목록에 들어있는 완충기페지의 수가 다음 식의 림계값을 넘었음을 확인했을 때

p * bdf_prm. b_un. nfract_sync / 100

여기서 p는 체계에 있는 완충기페지로 사용될수 있는 페지의 수이다.(기본적으로 이 값은 DMA와 일반기억기지역(zone)에 있는 모든 페지의 수이다.) 실제로 이 계산은 balance_dirty_state()함수에서 이루어지며 이 함수는 불결한 또는 잠근완충기가 nfract: 림계값보다 작으면 -1, nfract와 nfract_sync 사이이면 0, nfract_sync보다 크면 1을 반환한다. balance_dirty() 함수는 보통 완충기가 불결한것으로 표시되면 호출되며 이 함수는 완충기머리부를 BUF_DIRTY 목록으로 옮긴다.

・try_to_free_buffers() 함수가 일부완충기페지의 완충기머리부를 해제하는 작업을 실패했을 때(《완충기페지할당》 참고)

・grow_buffers()함수가 새로운 완충기페지할당에 실패했을 때 또는 create_buffers() 함수가 새로운 완충기머리부 할당에 실패했을 때(《완충기페지할당》 참고)

・사용자가 콘솔의 특정한 건조합을 눌렀을 때.(보통 ALT+sysRq+U와 ALT_sysRq+s) 이 건조합은 Linux핵심부에서 《Magic SysRq Key》항목을 선택하여 콤파일했을 때에만 사용할수 있으며 Linux핵심부해커가 일부핵심부동작을 직접 조종할수 있도록 한다.

bdflush를 깨우기 위해 핵심부는 wakeup_bdflush() 함수를 호출한다. 이 함수는 단순히 다음을 실행하여 bdflush_wait 작업대기렬에서 잠들어있는 처리기를 깨운다.

wake_up_interruptible(&bdflush_wait);

이 대기렬에는 bdflush 처리기 하나만 있다.

bdflush()함수의 핵심은 다음과 같은 무한순환이다.

```
for (;;){
if (emergenoy_sync_scheduled) /* Magic syqRq Key 지원을 포함하여 */
do_emergency_sync( ); /* 핵심부를 콤파일했을 때 */
spin_1ock(&lru_list_lock);
if (!write_some_buffers (0) || balance_dirty_state ( ) < 0) {
wait_for_some_buffers (0);
interruptible_sleep_on (&bdflush_wait);
}
}
```

Magic SysRq Key 항목을 선택하여 콤파일한 Linux핵심부의 경우 bdflush()는 사용자가 긴급동기화를 요청했는지 검사하고 요청이 있었으면 do_emergency_sync()를 호출하여 모든 블로크장치에 대해 fsync_dev()를 호출하여 모든 불결한 완충기를 흘리기한다.(뒤에 나오는 《sync(), fsync() 와 flatasync() 체계호출》 참고)

다음으로 이 함수는 lru_list_lock 스핀잠그기를 얻은 다음 write_some_buffers() 함수를 호출하여 잠그기가 걸리지 않은 불결한 완충기 32개까지에 대해 블로크입출력쓰기연산의 활성화를 시도한다. 쓰기연산이 활성화되면 write_some_buffers()는 lru_list_lock 스핀잠그기를 해제하고 잠그기되지 않은 불결한 완충기를 찾은 수가 32개 미만인 경우 0을 반환하고 그렇지 않으면 부수값을 반환한다.

write_some_buffers()가 흘리기할 완충기 32개를 찾지 못했거나 불결한 혹은 잠그기걸린 완충기의 수가 bdflush의 변수 nfract의 비률미만으로 내려가면 bdflush 핵심부스레드는 잠든다. 이를 위해 먼저 wait_for_some_buffers()함수를 호출하여 BUF_LOCKED 목록의 모든 입출력자료전송이 끝날 때까지 잠든다. 이 시간동안 핵심부스레드는 핵심부가 wakeup_bdflush()함수를 호출할지라도 깨여나지 않는다. 자료전송이 끝나면 bdflush()함수는 bdflush_wait대기렬에 대해 interruptible_sleep_on()을 호출하여 다음에 wakeup_bdflush()가 다시 호출될 때까지 잠든다.

○ **kupdate핵심부스레드**

불결한 완충기가 너무 많을 때 또는 완충기가 더 필요한데 사용할수 있는 기억기가 부족할 때 bdflush 핵심부스레드가 활성화되므로 어떤 불결한 완충기는 기억기에 필요이상으로 오래 머문 다음 디스크에 흘리기된다. 따라서 너무 오래된 불결한 완충기를 흘리기 위해 kupdate 핵심부스레드를 도입하였다.

표 4-11에서 살펴본것처럼 age_buffer는 kupdate가 완충기를 디스크에 기록할 때까지의 시간(보통 30s)이고 bdf_prm표의 interval마당은 kupdate 핵심부스레드를 활성화하는 시간간격을 틱크단위로 나타낸 값(보통 5s)이다. 이 마당이 0이면 SIGCONT 신호를 받을 때까지 핵심부스레드가 중단된다.

핵심부가 완충기의 내용을 변경했을 때 완충기에 대응하는 완충기머리부의 b_flushtime 마당을 후에 디스크에 완충기를 흘리기할 시간(jiffies단위)으로 설정한다. kupdate 핵심부스레드는 b_flushtime마당값이 jiffies의 현재값보다 작은 불결한 완충기만 선택하여 흘리기한다.

kupdate핵심부스레드는 kupdate() 함수를 호출하며 이 함수는 wb_kupdate()함수를 호출하는데 다음과 같은 순환을 실행한다.

```
while (nr_to_write > 0) {
        wbc.encountered_congestion = 0;
        wbc.nr_to_write = MAX_WRITEBACK_PAGES;
        writeback_inodes(&wbc);
        if (wbc.nr_to_write > 0) {
                if (wbc.encountered_congestion)
                        blk_congestion_wait(WRITE, HZ/10);
                else
                        break;
        }
        nr_to_write -= MAX_WRITEBACK_PAGES - wbc.nr_to_write;
}
```

먼저 핵심부스레드는 BUF_LOCKED 목록의 모든 완충기에 대한 입출력자료전송이 끝날 때까지 자신을 연기한다. 그리고 bdf.prm. b_un.interval이 0이 아니면 스레드는 지정한 틱크만큼 잠든다.(《 동적시간응용 》 참고). 그렇지 않으면 스레드는 SIGCONT신호를 받을 때까지 자신을 중단한다.(《신호의 역할》 참고)

kupdate()함수의 핵심은 sync_old_buffers() 함수이다. 이 함수가 실행하는 연산은 Unix에서 사용하는 표준 파일체계의 경우 매우 간단하다. 함수가 수행하는 일은 불결한 완충기를 디스크에 기록하는것이 전부이다. 그렇지만 표준파일체계가 아닌 경우 초블로크나 i마디정보를 복잡한 방식으로 저장하므로 복잡한 문제가 발생할수 있다.

sync_old_buffefs()는 다음과 같은 단계를 실행한다.

1. 대형(big) 핵심부잠그기를 획득한다.

2. sync_unlocked_inodes()를 호출하여 탑재된 모든 파일체계의 초블로크를 탐색하고 각 초블로크에 대해 초블로크객체의 s_dirty 마당이 가리키는 불결한 색인마디들을 탐색한다. 이 함수는 각 색인마디에 대응하는 파일의 기억기배치에 속한 불결한 페지들을 흘리기한다.(《 불결한 기억기배치페지를 디스크에 흘리기 》 참고) 그리고 write_inode 초블로크 연산이 정의되여있다면 호출한다.(write_inode메쏘드는 모든 색인마디자료를 디스크블로크 하나에 저장하지 않는 비Unix계렬 파일체계, 례를 들면 MS_DOS 파일체계에서만 정의되여있다.)

3. sync_supers()를 호출한다. 이 함수는 모든 초블로크자료를 디스크블로크 하나에 저장하지 않는 파일체계(례를 들면 HFS)에서 사용하는 초블로크를 관리한다. 이 함수는 현재 탑재된 모든 파일체계의 초블로크를 탐색하여(《파일체계탑재》 참고) 각 초블로크에 대해 write_super초블로크연산이 정의되여있다면 실행한다.(《초블로크객체》 참고) Unix계렬 파일체계에서는 write_super 메쏘드를 정의하지 않는다.

4. 큰 핵심부잠그기를 해제한다.

5. 다음으로 구성된 순환을 시작한다.

a. lru_list_lock 스핀잠그기를 획득한다.

b. BUF_DIRTY 목록의 첫번째 완충기머리부를 가리키는 지시자 bh를 얻는다.

c. 지시자가 NULL이거나 완충기머리부의 마당 bflushtlme 값이 jiffies보다 크면 (오래되지 않은 완충기) lru_list_lock 스핀잠그기를 해제하고 완료한다.

d. write_some_buffers()를 호출하여 BUF_DIRTY 목록의 잠그기되지 않은 불결한 완충기 32개까지에 대해 블로크입출력 쓰기활성화를 시도한다. 쓰기활성화를 실행하면 write_some_buffers()는 lru_list_lock 스핀잠그기를 해제하고 잠그기가 걸리지 않은 불결한 완충기를 찾은 수가 32개 미만이면 0을 반환하며 그렇지 않으면 부수값을 반환한다.

e. write_some_buffers()가 디스크에 흘리기한 잠그기하지 않은 완충기수가 32이면 5a단계로 이동한다. 그렇지 않으면 실행을 완료한다.

◦ sync(), fsync()와 fdatasync() **체계호출**

사용자응용프로그람은 불결한 완충기를 디스크에 흘리기 위해 다음 3가지 체계호출을 사용할수 있다.

sync()

보통 체계끄기직전에 호출한다. 모든 불결한 완충기를 디스크에 흘린다.

fsyncO

처리기가 현재 열려있는 특정한 파일에 속한 모든 블로크를 디스크에 흘리도록 한다.

fdatasync()

fdatasync()와 매우 류사하지만 파일의 색인마디 블로크를 흘리지 않는다.

sync()체계호출의 핵심은 fsync_dev()함수로 다음과 같은 작업을 수행한다.

1. sync_buffers()를 호출하여 다음 코드를 실행한다.

```
do {
spin_lock(&lru_list_lock);
} while (write_some_buffers(0));
run_task_queue (&tq_disk);
```

이 함수는 잠그기걸리지 않은 불결한 완충기를 32개 찾을 때까지 write_some_buffers()함수를 호출한다. 그리고 블로크장치구동프로그람이 실제 입출력

자료전송을 시작하도록 한다.

2. 큰 핵심부잠그기를 획득한다.

3. sync_inodes()를 호출한다. 이 함수는 앞서 설명한 sync_unlocked_inodes와 아주 류사하다.

4. sync_supers()를 호출하여 불결한 초블로크를 디스크에 기록한다. 필요하다면 write_super 메쏘드를 호출한다.

5. 큰 핵심부잠그기를 해제한다.

6. sync_buffers()를 다시 한번 호출한다. 이번에는 잠그지 않은 모든 완충기의 전송이 끝날 때까지 기다린다.

fsync()체계호출은 핵심부가 fd전 파일서술자변수가 지정하는 파일에 속한 모든 불결한 완충기를 디스크에 기록하게 한다.(필요하다면 색인마디를 소유하고있는 완충기를 포함한다.) 체계봉사루틴은 파일객체의 주소를 얻어서 파일객체의 fsync 메쏘드를 호출한다. 일반적으로 이 메쏘드는 단순히 fsync_inode_buffers() 함수를 호출한다. 이 함수는 색인마디객체의 불결한 완충기목록 두개를 탐색하여(《캐쉬완충기에 대한 완충기머리부목록》 참고) 목록의 각 입구점에 대해 ll_rw_block()를 호출한다. 이 함수는 잠근 완충기에 대해 wait_on_buffer()를 호출하여 파일의 모든 불결한 완충기가 디스크에 기록될 때까지 함수를 호출한 처리기를 연기한다. 그리고 fsync()체계호출의 봉사루틴은 파일의 기억기배치에 속한 불결한 페지가 있다면 이것들을 흘린다.(《불결한 기억기배치페지를 디스크에 흘리기》 참고)

fdatasync()체계호출은 fsync()와 매우 류사하지만 색인마디정보를 포함한 완충기는 기록하지 않고 파일자료를 포함한 완충기만 기록한다. Linux 2.6은 fdatasync()를 위한 별도의 파일메쏘드를 제공하지 않으며 이 체계호출은 fsync메쏘드를 사용한다. 따라서 fdatasync()는 fsync()와 동일하다.

제 4 절. 교환

앞에서 디스크캐쉬를 살펴보았다. 디스크캐쉬는 RAM을 디스크의 확장으로 사용하며 디스크접근회수를 줄여서 체계의 응답시간을 향상하는것이 목적이다. 이 절에서 소개하는 교환(swapping)기법의 목적은 정반대이다. 핵심부는 디스크공간의 일부를 RAM의 확장으로 사용한다. 프로그람작성자는 교환의 존재를 알지 못한다. 교환령역을 설정하여 활성화하면 프로쎄스는 자신이 지정하는 주소들이 모두 물리적인 기억기를 사용하고있다고 가정하고 동작하며 핵심부가 페지의 일부를 디스크에 저장하였다가 필요할 때 다시 가져온다는 사실을 알지 못한다. 디스크캐쉬는 여유RAM을 사용하여 체계성능을 향상하는 반면에 교환은 기억기용량을 확장하기 위해 기억기접근속도를 떨군다. 따라서

디스크캐쉬는 좋은것이지만 교환은 여유RAM이 너무 모자랄 때 사용하는 최후의 수단이라고 생각하는것이 좋다. 이 절은《교환이란 무엇인가》에서 교환을 정의하는것부터 시작한다. 교환령역절에서는 교환을 실현하기 위한 주요자료구조를 설명한다. 그리고 교환캐쉬와 RAM과 교환령역사이에서 페지를 전송하는 저수준함수를 설명한다. 다음 두 부분은 특히 중요하다.《페지를 교환하여 내보내기》에서는 디스크에 교환하여 내보낼 페지를 선택하는 기법을 설명하고《페지교환하여 끌여넣기》에서는 필요할 때 교환령역에 저장된 페지를 RALM으로 다시 읽어들이는 기법을 설명한다. 교환캐쉬를 포함하여 여러 종류의 디스크캐쉬가 있기때문에 이 캐쉬들이 사용가능한 모든 RAM을 사용하게 되여 여유RAM이 없게 된다. 이것을 해결하기 위해 핵심부가 어떻게 여유RAM용량을 감시하고 필요할 때 캐쉬 또는 프로쎄스주소공간에서 페지를 해제하여 여유기억기를 확보하는지 살펴본다.

1. 교환이란 무엇인가

교환의 목적은 다음 두가지이다.

· 프로쎄스가 실제로 사용할수 있는 주소공간확장

· 프로쎄스를 적재할 동적RAM공간(핵심부코드와 정적자료구조를 초기화하고 남은 RAM공간)의 확장

교환이 사용자에게 어떤 도움을 주는지 몇가지 례를 들어보자. 가장 단순한 례는 프로그람의 자료구조가 실제 RAM크기보다 더 많은 공간을 차지하는 경우이다. 교환령역은 문제없이 이 프로그람을 적재하여 실행할수 있게 해준다. 좀 더 특이한 실례로서는 기억기를 많이 요구하는 큰 응용프로그람 몇개를 동시에 실행하도록 사용자가 여러 명령을 내린 경우를 들수 있다. 교환령역이 없으면 체계는 새로운 응용프로그람 실행요청을 거부할수도 있다. 반면에 교환령역은 이미 실행중인 프로쎄스를 제거하지 않고 그중 일부기억기만 해제하여 핵심부가 새로운 응용프로그람을 실행할수 있게 한다.

우의 두 실례에서 교환의 우점과 함께 부족점도 알수 있다. 가상적으로는 RAM이 더 있는것처럼 보이지만 성능면에서는 실제 RAM을 따라갈수 없다. 프로쎄스가 나간 페지에 접근하는데 걸리는 시간은 RAM에 비해 수백배나 느리다. 간단히 말해서 성능이 중요하다면 교환은 최후의 수단이다. 늘어나는 처리요구에 대응하는 가장 좋은 해결방법은 여전히 RAM소편을 더 추가하는것이다. 그러나 체계의 전체적인 측면에서 교환이 효과적인 경우도 있다. 오래동안 실행되는 프로쎄스는 보통 자신이 얻은 페지들의 절반정도에만 접근한다. 사용가능한 RAM이 있을 때에도 사용되지 않은 페지를 교환하여 내보내고 해당 RAM을 디스크캐쉬로 사용하는것이 전체적인 체계성능을 높이는 경우가 있다.

교환은 고전적인 기법이다. 초기 Unix체계핵심부에서는 여유기억기용량을 계속해서 감시하다가 여유 기억기가 고정된 림계값보다 적어지면 교환하여 내보내기를 실행하

였다. 이때 프로쎄스의 전체 주소공간을 디스크에 복사하였다. 반대로 순서짜기알고리듬이 교환하여 내보낸 프로쎄스를 실행하도록 선택하면 프로쎄스전체를 디스크에서 교환하여 넣었다.

Linux를 비롯한 최신 Unix 핵심부에서는 이런 접근방법을 사용하지 않는다. 가장 큰 리유는 교환하여 내보낸 프로쎄스전체를 교환하여 넣으려면 프로쎄스의 문맥전환비용이 너무 높아지기때문이다. 이런 교환작업의 부담을 보상하려면 순서짜기알고리듬이 매우 복잡해진다. 즉 RAM에 있는 프로쎄스를 선취하면서도 교환하여 내보낸 프로쎄스를 완전히 무시해서는 안된다.

Linux의 교환은 프로쎄스주소공간 전체가 아닌 각 페지단위를 대상으로 이루어진다. CPU에 포함된 하드웨어페지화장치에 의해 이렇게 작은 단위를 대상으로 교환을 수행할수 있다. 《정규페지화》를 보면 각 페지표입구점 (PTE)에는 present기발이 있다. 핵심부는 이 기발을 활용하여 프로쎄스주소공간의 페지가 교환하여 내보냈다는 사실을 하드웨어에 알릴수 있다. Linux는 이 기발외에 페지표입구점의 남은 비트를 활용하여 교환하여 내보낸 페지가 디스크에 저장된 위치를 저장한다.

《페지오유례외(exception)》가 발생하면 례외조종기가 RAM에 해당 페지가 없다는 사실을 알게되며 디스크에서 해당 페지를 교환하여 넣는 함수를 호출한다.

교환에서 복잡한 부분은 주로 교환하여 내보낸 부분이며 특히 4가지 주요문제를 고려해야 한다.

.어떤 종류의 페지를 교환하여 내보낼것인가.

.교환령역안에서 페지를 어떻게 분산할것인가.

.교환하여 내보낼 페지를 어떻게 선택할것인가.

.언제 페지를 교환하여 내보낼것인가.

Linux가 이 4가지 문제를 어떻게 처리하는지 간단히 살펴본 다음 교환과 관련있는 주요 자료구조와 함수를 보도록 한다.

1) 교환하여 내보낼 페지종류

교환은 다음 종류의 페지에만 적용할수 있다.

- 프로쎄스의 닉명 기억기령역에 속한 페지(례를 들면 사용자방식탄창)
- 프로쎄스의 비공개(private)기억기배치에 속한 변경된 페지
- IPC공유기억기령역에 속한 페지(《IPC 공유기억기》참고)

(교환하여 내보낼 대상이 아닌)나머지 종류의 페지는 핵심부에서 사용하거나 디스크상에 있는 파일을 배치할 때 사용하는 페지이다. 핵심부가 사용하는 페지는 교환이 무시한다. 이렇게 해야 핵심부설계를 단순하게 할수 있기때문이다. 파일배치에 사용하는 페지의 경우에 가장 좋은 교환령역은 해당 파일 그자체이다.

2) 교환령역안의 페지분산방법

각 교환령역은 슬로트로 구성되며 각 슬로트는 한 페지를 저장한다. 교환하여 내보내는 경우 핵심부는 후에 교환령역에 접근할 때 디스크탐색시간을 최소화하기 위해 련속하는 슬로트에 페지를 저장하려 한다. 이것은 효률적인 교환알고리듬에 중요한 요소이다.

교환령역을 두개 이상 사용하는 경우 문제가 더 복잡해진다. 빠른 교환령역 즉 빠른 디스크에 저장된 교환령역이 높은 우선순위를 가진다. 여유슬로트를 찾을 때 우선순위가 가장 높은 교환령역부터 탐색을 시작한다. 우선순위가 같은 교환령역이 여러개 있으면 그중 하나에 부하가 집중되는것을 막기 위해 번갈아 가면서 선택한다. 우선순위가 가장 높은 교환령역에 여유슬로트가 없으면 그 다음 우선순위의 교환령역에서 탐색을 계속한다.

3) 교환하여 내보낼 페지선택방법

교환하여 내보낼 페지를 선택할 때 특정기준에 따라 페지에 순서를 매길수 있으면 좋을것이다. 그래서 여러 조작체계핵심부에서 몇가지 LRU(Least Recently Used) 교체알고리듬을 제안하고 사용해왔다. 핵심사상은 RAM의 각 페지에 페지의 나이(age), 즉 해당 페지에 마지막으로 접근한 이후 경과한 시간을 저장해두는것이다. 그리고 프로쎄스에서 가장 오래된 페지를 교환하여 내보낸다.

어떤 콤퓨터기반에서는 LRU알고리듬을 위해 복잡한 지원을 제공한다. 례를 들어 어떤 주프레임의 CPU는 각 페지표입구점의 페지나 이를 나타내는 계수기값을 자동으로 갱신한다. 그러나 80x86처리기는 그런 하드웨어기능을 제공하지 않으므로 Linux는 진정한 LRU알고리듬을 사용할수 없다. 대신 Linux에서는 페지에 접근할 때마다 하드웨어가 자동으로 설정하는 각 페지표입구점의 Accessed기발을 활용하여 교환하여 내보낼 후보를 선택한다. 후에 보지만 페지를 자주 교환하여 넣기/내보내지 않도록 약간 단순한 방법을 사용하여 이 기발을 설정하거나 지운다.

페지를 교환하여 내보낼 때 핵심부가 사용할수 있는 기억기가 위험할 정도로 모자랄 때 효과적이다. 심하게 기억기가 모자라는 경우에 대비한 긴급수단으로 핵심부는 여유페지를 몇개를 예약해둔다. 이것들은 가장 핵심적인 함수들만 사용할수 있다. 이 여유페지는 체계파괴를 막기 위한 핵심적인 기능이다. 여유페지가 없다면 자원을 해제하기 위해 호출한 핵심부루틴이 작업을 정상적으로 완료하는데 필요한 기억기령역을 얻지 못하는 경우가 발생하기때문이다. Linux는 여유페지들을 보호하기 위하여 다음과 같은 경우에 교환하여 내보내기를 실행한다.

· 여유페지들의 수가 미리 정의된 림계값보다 작아질 때 마다 주기적으로 활성화되는 kswapd 핵심부스레드에 의해 실행

· 여유페지들의 수가 미리 정의된 림계보다 작아져 형제체계(《형제체계 알고리듬》참고)에 대한 기억기요청을 만족하지 못하는 경우

2. 교환령역

교환령역(swap area)은 기억기로부터 교환하여 내보낸 페지를 저장하는 공간이다.

교환령역은 독자적인 디스크구획일수도 있고 구획에 포함된 한개의 파일일수도 있다. 여러 교환령역을 정의할수 있으며 교환령역의 최대 개수는 MAX_SWAPFILES마크로(보통 32로설정)로 지정한다. 체계관리자는 여러 교환령역을 사용하여 교환공간을 여러 디스크에 분산해서 하드웨어가 이것들에 대해 동시에 병렬적으로 작업할수 있게 한다.

또한 체계를 재시동하지 않고 실행시간에 교환령역을 늘일수 있게 한다.

각 교환령역은 련속된 페지슬로트들로 이루어진다. 교환하여 내보낸 페지를 저장하기 위해 4096B블로크를 사용한다. 교환령역의 첫번째 페지슬로트는 교환령역에 대한 정보를 저장하는데 사용한다. 첫번째 슬로트의 자료구조는 info와 magic 두 구조체가 들어있는 swap_header이다. magic구조체는 디스크의 특정한 부분이 교환령역임을 나타내는 문자렬을 제공한다. 이 구조체는 magic.magic이라는 마당 하나로 이루어져있으며 문자 10개로 이루어진 특정한 문자렬을 저장한다. magic 구조체의 핵심적인 역할은 핵심부가 어떤 파일이나 구획이 교환령역임을 정확히 알수 있도록 하는것이다. 이 문자렬의 실제값은 교환알고리듬판본에 따라 다르며 판본 1인 경우 SWAP_SPACE, 판본 2인 경우 SWAPSPACE2이다. 이 마당의 위치는 언제나 첫번째 페지슬로트의 맨 끝부분이다.

info구조체의 마당은 다음과 같다.

info.bootbits

교환알고리듬에서 사용하지 않는다. 이 마당은 교환령역의 처음 1024B이며 구획자료, 디스크표식 등의 정보를 저장한다.

info.version

교환알고리듬의 판본

info.last_page

사용가능한 마지막페지슬로트

infonr_badpages

결함이 있는 페지슬로트의 수

info.padding[125]

메꾸기바이트

info.badpages[1]

결함이 있는 페지슬로트의 위치를 나타내는 637개까지의 수

교환령역의 자료는 체계가 켜져있는 동안에만 유효하다. 체계가 중지될 때 모든 프로쎄스를 제거하며 프로쎄스가 교환령역에 저장했던 자료도 삭제된다. 이런 리유로 교환령역이 담고있는 조종정보는 교환령역형과 결함이 있는 페지슬로트목록정보이다. 조종정보는 4kB페지하나에 쉽게 저장할수 있다.

일반적으로 체계관리지는 Linux체계의 다른 구획을 생성할 때 교환구획도 같이 생성하고 /sbin/mkswap 명령을 사용하여 이 디스크령역을 새로운 교환령역으로 설정한다. 이 명령은 앞서 설명한것처럼 첫번째 페지슬로트의 마당을 설정한다. 디스크가 결함이 있는 블로크를 포함하고있을수 있으므로 이 프로그람은 결함이 있는 블로크를 찾기 위해 다른 페지의 모든 슬로트를 검사한다. 그러나 /sbin/mkswap 명령은 교환령역을

활성화하지 않은채로 남겨둔다. 각 교환령역은 체계를 시동하는 파정에서 스크립트파일로 활성화하거나 체계가 동작하는중에 동적으로 활성화할수 있다. 초기화한 교환령역은 실제로 체계RAM의 확장령역을 나타내야만 활성화한 상태로 간주된다.(뒤에 나오는 《교환령역활성화와 비활성화》 참고)

1) **교환령역서술자**

활성화된 각 교환령역은 자신의 swap_info_struct서술자를 기억기에 저장한다. 서술자의 마당은 표 4-12와 같다.

표 4-12. 교환령역서술자마당

형	마 당	설 명
unsigned int	flags	교환령역기발
kdev_t	swap_device	교환디스크구획의 장치번호
spinlock_t	sdev_lock	교환령역서술자스핀잠그기
struct dentry*	swap_file	파일 또는 장치파일의 덴트리
struct vfsmount*	swap_vfsmnt	파일 또는 장치파일의 탑재된 파일체계서술자
unsigned short*	swap_map	각 교환령역페지의 슬로트마다 하나씩 할당되는 계수기배렬에 대한 지적자
unsigned int	lowest_bit	여유슬로트를 찾을 때 살펴볼 처음페지슬로트
unsigned int	highest_bit	여유슬로트를 찾을 때 살펴볼 마지막페지슬로트
unsigned int	cluster_next	여유슬로트를 찾을 때 살펴볼 다음페지슬로트
unsigned int	cluster_nr	여유페지슬로트할당수. 이 수를 넘으면 처음부터 다시 시작한다.
Int	prio	교환령역우선순위
Int	pages	사용가능한 페지슬로트의 수
unsigned long	max	교환령역크기(페지단위)
Int	next	다음교환령역서술자의 지적자

flags 마당에는 서로 겹치는 두 보조마당이 있다.

SWP_USED

활성화한 교환령역이면 1, 활성화하지 않은 교환령역이면 0이다.

SWP_WR1TEOK

교환령역에 쓰기가 가능하면 이 두 비트마당을 3으로 설정하고 그렇지 않으면 0으로 설정한다. 이 마당의 아래비트는 SWP_USED비트로 사용하는 비트와 겹치기때문에 교환령역을 활성화했을 때에만 쓰기가 가능하다. 핵심부는 활성화 또는 비활성화도중에는 교환령역에 쓰기를 허용하지 않는다.

swap_map마당은 각 교환령역의 페지슬로트에 대한 계수기배렬을 가리킨다. 계수기가 0이면 해당 페지슬로트는 비여있다. 계수기가 정수면 페지슬로트는 교환하여 내보낸 페지로 채워져있다.(정수값의 정확한 의미는 《교환캐쉬》에서 설명한다.) 계수기값이 SWAP_MAP_MAX(값은 32 767)이면 페지슬로트에 저장된 페지는 《영구적(permanent)》이며, 슬로트에서 제거할수 없다. 계수기값이 SWAP_MAP_BAD(값은 32 768)이 면 페지슬로트에 결함이 있는것이므로 사용할수 없다.

prio마당은 교환 보조체계가 각 교환령역을 사용하는 우선순위를 나타내는 부호있는 정수이다. 고속의 디스크에 있는 교환령역은 먼저 사용되도록 우선순위가 높아야 한다. 우선순위가 높은 교환령역을 모두 사용하면 교환 알고리듬은 낮은 수준의 교환령역을 사용한다. 우선순위가 같은 교환령역의 경우 교환하여 내보낸 페지를 분산하도록 번갈아가면서 선택한다. 《교환령역 활성화와 비활성화》에서 보지만 우선순위할당은 교환령역을 활성화할 때 이루어진다. sdev_lock마당은 SMP체계에서 서술자에 동시에 접근하는것을 방지하기 위한 스핀잠그기이다.

swap_info배렬은 교환령역서술자 MAX_SWAPFILES개를 포함한다. 물론 이것들을 모두 사용하지는 않으며 SWP_USED기발을 설정한 서술자만 사용한다. 그림 4-21은 swap_info배렬과 교환령역 그리고 여기에 대응하는 계수기배렬을 보여준다.

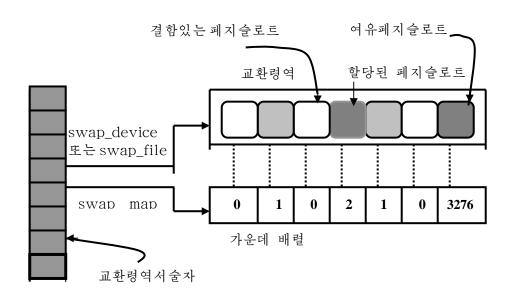



그림 4-21. 교환령역의 자료구조

nr_swapfiles는 사용된 교환령역서술자를 포함한적이 있는 마지막 배렬항목의 색인값을 저장한다. 이름이 나타내는것과 달리 이 변수는 활성상태인 교환령역의 수를 저장

하지 않는다. 활성상태인 교환령역의 서술자는 우선순위에 따라 정렬된 목록에 들어간다. 이 목록은 교환령역서술자의 next 마당으로 실현되며 이 마당에는 다음 서술자의 swap_info배렬에서의 색인값을 저장한다. 지금까지 next라는 이름은 주로 지적자를 나타냈는데 여기서는 색인값으로 사용한다.

swap_list변수는 swap_list_t형이고 다음과 같은 마당을 포함한다.

head

첫번째 목록요소가 swap_info배렬에서 위치한 색인값.

next

다음 페지를 교환하여 내보내기 위해 사용할 교환령역의 서술자가 swap_info배렬에서 위치한 색인값.

이 마당은 여유슬로트가 있는 가장 높은 우선순위의 교환령역중에서 원형(round_robin, 돌아가면서 하나씩 선택하는 간단한 알고리듬)알고리듬을 실현한다.

swaplock스핀잠그기는 다중처리기체계에서 목록에 동시에 접근하는것을 방지하기 위한것이다. 교환령역서술자의 max마당은 교환령역크기를 페지단위로 저장하는 반면에 pages마당은 사용가능한 페지슬로트의 수를 저장한다. pages는 첫번째 페지슬로트와 결함이 있는 페지슬로트를 고려하지 않기때문에 이 수자는 서로 다르다.

끝으로 nr_swap_pages변수는 모든 활성화된 교환령역중에서 사용가능한 페지슬로트(결함이 없는 여유페지슬로트)의 수를 저장하고 total_swap_pages는 결함이 없는 페지슬로트의 수를 저장한다.

2) 교환하여 내보낸 페지식별자

교환하여 내보낸 페지는 swap_info배렬에서 교환령역의 색인값과 해당 교환령역안에서 페지슬로트의 색인값을 통해 아주 쉽게 식별할수 있다. 교환령역의 첫번째 페지(색인값 0)는 앞서 설명한 swap_header를 위해 예약되여있으므로 사용할수 있는 첫번째 페지슬로트의 색인값은 1이다. 교환하여 내보낸 페지식별자의 형식은 그림 4-22와 같다.

31 ... 8	7 ... 1	0
페지슬로트색인값	령역번호	0

그림 4-22. 교환하여 내보낸 페지식별자

SWP_ENTRY(type,offset)마크로는 교환령역색인값 type와 페지슬로트색인값 offset로부터 교환하여 내보낸 페지식별자를 생성한다. 반대로 SWP_TYPE와 SWP_OFFSET마크로는 교환하여 내보낸 페지식별자로부터 교환령역색인값과 페지슬로

교육성 교육정보쎈터

트색인값을 뽑아낸다.

페지를 교환하여 내보내면 해당 페지의 식별자를 페지표에 페지입구점으로 삽입하여 필요할 때 찾을수 있게 한다. 이 식별자의 제일 아래비트는 Present 기발에 대응하는데 언제나 0으로 설정되여 이 페지가 현재 RAM에 없음을 나타낸다. 그러나 웃자리 30bit 중에서 적어도 한 비트는 1이여야 한다.

교환령역 0의 슬로트 0에는 저장되는 페지가 없기때문이다. 따라서 페지표 입구점 값으로부터 다음과 같은 3가지 경우를 구분할수 있다.

o **빈 입구점**

이 페지는 프로쎄스의 주소공간에 속하지 않는다(모든 비트가 0인 경우).

웃자리 31bit중에 1인 비트가 있고 마지막 비트가 0인 페지는 현재 교환하여 내보냈다.

o **제일아래자리비트가 1**

이 페지는 RAM에 있다.

교환령역의 최대크기는 슬로트를 나타내는데 사용하는 비트의 수에 따라 결정된다. 80x86방식에서 이 비트의 수는 24bit이므로 교환령역의 최대크기는 224개 슬로트가 된다.(슬로트크기가 4kB므로 64GB까지 가능하다.)

한 페지가 여러 프로쎄스의 주소공간에 속할수 있으므로(《교환캐쉬》 참고) 페지는 어떤 프로쎄스의 주소공간에서 교환하여 내보내기되였어도 여전히 주기억기에 있을수 있으며 따라서 한 페지를 여러번 교환하여 내보낼수 있다. 물론 페지는 물리적으로 한번만 교환하여 내보내여 저장되며 처음 교환하여 내보낼 때에도 이후에는 swap_map계수기만 증가한다.

이미 교환하여 내보낸 페지를 다시 교환하여 내보내면 swap_duplicate()함수를 호출한다. 이 함수는 변수로 받은 교환하여 내보낸 페지식별자가 유효한지 확인하고 대응하는 swap_map계수기를 증가시킨다. 좀 더 자세히 설명하면 다음과 같은 작업을 수행한다.

1. SWP_TYPE와 SWP_OFFSET마크로를 사용하여 변수에서 구획번호 TYPE와 페지슬로트색인값 offset를 뽑아낸다.

2, 활성화된 교환령역인지 검사한다. 그렇지 않으면 0을 반환한다.(잘못된 식별자)

3. 페지슬로트가 유효하고 내용이 있는것인지 검사한다.(swap_map계수기가 0보다 크고 SWAP_MAX_BAD보다 작은 경우) 그렇지 않으면 0을 반환한다.(잘못된 식별자)

4. 교환하여 내보낸 페지식별자는 유효한 페지를 가리킨다. swap_map계수기가 SWAP_MP_MAX에 도달하지 않았으면 1 증가시킨다.

5. 1을 반환한다.(유효한 식별자)

3) 교환령역활성화와 비활성화

교환령역을 초기화하면 초사용자(좀 더 정확히 말하자면 《프로쎄스의 자격과 특질》

에서 설명하는 CAP_SYS_ADMIN권한을 소유한 모든 사용자)는 /bin/swapon과 /bin/swapoff를 사용하여 교환령역을 활성화하거나 비활성화 할수있다. 이것들은 swapon()과 swapoffO체계호출을 사용한다. 이것들에 대응하는 봉사루틴을 간단히 보자.

◦ sys_swapon( )봉사루틴

sys_swapon()봉사루틴은 다음과 같은 변수를 받는다.

specialfile

이 변수는 장치파일(구획) 또는 교환령역실현에 사용한 일반파일의 경로명 (사용자방식주소공간에 있다.)을 나타낸다.

swap_flag

이 변수는 SWAP_FLAG_PREFER 한 비트와 교환령역의 우선순위를 나타내는 31bit로 구성되여있다.(15bit는 SWAP_FLAG_PREFER가 설정되여있을 때에만 의미가 있다.)

이 함수는 교환령역이 생성될 때 첫번째 슬로트에 들어간 swap_header의 마당을 검사한다. 이 함수가 실행하는 주요 단계는 다음과 같다.

1. 현재프로쎄스가 CAP_SYSYSYS_ADMIN 권한이 있는지 검사한다.

2. 교환령역서술자의 배렬 swap_info에서 SWP_USED기발이 설정되지 않은 첫번째 서술자 즉 대응하는 교환령역이 활성화되지 않은 서술자를 찾는다. 찾지 못하였다면 이미 활성화된 교환령역 MAX_SWAPFILES개가 있다는 의미이므로 오유코드를 반환한다.

3. 교환령역에 대한 서술자를 찾았다. 함수는 서술자의 마당을 설정한다. (flags를 SWP_USED로 1owest_bit와 highest_ b i t 를 0으로) 그리고 서술자의 색인값이 nr_swapfiles보다 크면 이 변수(nr_swapfiles)를 변경한다.

4. swap_ffflags변수에 새로운 교환령역의 우선순위가 지정되여있으면 함수는 서술자의 prio마당을 설정한다. 그렇지 않으면 이 마당을 모든 활성화된 교환령역의 우선순위중 가장 작은 값보다 1작은 값으로 설정한다.(가장 후에 활성화한 교환령역이 가장 느린 블로크장치에 있다고 가정한다.) 이미 활성화 다른 교환령역이 없으면 1로 설정한다.

5. specialfile변수가 가리키는 문자렬을 사용자방식주소공간에서 복사한다.

6. 사용자방식주소공간에서 복사한 문자렬에 대서 경로명탐색을 하기 위해 path_init()와 path_walk()를 호출한다.

7. path_walk()가 반환한 등록부입구점객체와 태워진 파일체계서술자의 주소를 교환령역서술자의 sw_file과 swap_vfsmnt마당에 각각 저장한다.

8. specialfile변수가 블로크장치파일을 나타내면 함수는 다음과 같은 단계를 실행한다.

a. 장치번호를 서술자의 swap_device 마당에 저장한다.

b. 장치의 블로크크기를 4kB로 설정한다. 즉 blksize_size입구점을 PAGE_SIZE

로 설정한다.

c. bd_acquire()와 do_open()을 호출하여 블로크장치구동프로그람을 호출한다.(《블로크장치구동프로그람 초기화》참고)

9. swap_info에서 다른 교환령역의 address_space객체를 살펴봄으로써 이미 활성화된 교환령역인지 검사한다.(교환령역서술자의 주소 q에 대해 대응하는 address_space객체를 q->swap_file->d_inode->i_mapping에서 얻을수 있다.) 이미 활성화된 령역이라면 오유코드를 반환한다.

10. 페지틀을 할당하고 rw_swap_page_nolock()을 호출하여(뒤에 나오는 《교환페지전송》참고) 교환령역의 첫번째 페지에 저장된 swap_header를 읽는다.

11. 교환령역의 첫번째 페지의 마지막 10개 문자가 SWAP_SPACE 또는 SWAPSPACE2(교환알고리듬에는 약간 다른 두가지 판본이 있다.)와 일치하는지 검사한다. 일치하지 않으면 specialfile변수가 초기화한 교환령역이 아니므로 오유코드를 반환한다. 론의를 간단히 하기 위해 교환령역의 문자렬이 SWAPSPACE2이라고 가정한다.

12. 교환령역서술자의 1owest_bit, highest_bit마당을 swap_header의 info.last_page마당에 저장한 교환령역의 크기에 따라 초기화한다.

13. vmalloc()을 호출하여 새로운 교환령역에 대응하는 계수기배렬을 생성하고 그 주소를 교환서술자의 swap_map 마당에 저장한다. 이 배렬의 요소를 swap_header의 info.bad_pages마당에 저장되여있는 결함있는 페지슬로트목록에 따라 0 또는 SWAP_MAP_BAD로 초기화한다.

14. 첫번째 페지슬로트의 info.last_page와 info.nr_badpages마당값을 사용하여 사용가능한 페지슬로트의 수를 계산한다.

15. 교환서술자의 flags마당을 SWP_WRITEOK으로, pages마당을 사용가능한 페지슬로트의 수로 설정하고 nr_swap_pages와 total_swap_pages변수를 갱신한다.

16. 새로운 교환령역서술자를 swap_list변수가 가리키는 목록에 삽입한다.

17. 교환령역의 첫 페지의 자료를 포함하는 페지틀을 해제하고 0을 반환한다.(성공)

◦ sys_swapoff()봉사루틴

sys_swapoff()봉사루틴은 specialfile변수에 지정한 교환령역을 비활성(사용중단)상태로 만든다. 이 작업은 sys_swapon()에 비해 훨씬 더 복잡하고 시간이 많이 걸린다. 비활성화할 구획이 아직도 여러 프로쎄스에 속한 페지를 포함하고있을수 있기때문이다. 따라서 이 함수는 이 교환령역을 탐색하여 모든 페지를 교환하여 넣어야 한다. 각 교환하여넣기는 새로운 페지틀을 요구하기때문에 남아있는 페지틀이 더는 없을 경우 실패할수도 있다. 이 경우 함수는 오유코드를 반환한다. 작업은 다음과 같은 단계를 통해 이루어진다.

1. 현재프로쎄스에 CAP_SYS_ADMIN가 있는지 검사한다.

2. specialfile이 가리키는 문자렬을 복사하고 path_init()와 path_walk()를 호출

하여 경로명탐색을 실행한다.

3. swap_list가 가리키는 목록을 팀색하여 서술자의 swap_file 마당이 경로명탐색에서 찾은 등록부입구점객체를 가리키는 서술자를 찾는다. 이런 서술자가 없으면 잘못된 변수가 함수에 전달된것이므로 오유코드를 반환한다.

4. 그렇지 않고 서술자가 존재하면 서술자의 SWP_WRITEOK기발이 설정되여있는지 검사한다. 기발이 설정되여있지 않으면 교환령역을 이미 다른 프로쎄스가 비활성화한것이므로 오유코드를 반환한다.

5. 서술자를 목록에서 제거하고 flags마당을 SWP_USED로 설정하여 이 함수가 교환령역을 비활성화하기 전에 핵심부가 이 교환령역에 페지를 더 저장하지 않도록 한다.

6. 교환령역서술자의 pages 마당에 저장된 교환령역크기를 nr_swap_pages와 total_swap_pages 값에서 뺀다.

7. try_to_unuse()함수를 호출하여 교환령역에 남아있는 모든 페지를 RAM으로 가져오게 하고 이것들 페지를 사용하는 프로쎄스의 페지표를 갱신한다.

8. try_to_unuse()가 요청한 모든 페지들을 할당하는데 실패하면 교환령역을 비활성화할수 없다. 따라서 함수는 다음과 같은 단계를 실행한다.

a. 교환령역서술자를 다시 swap_list 목록에 넣고 flags마당을 SWP_WRITEOK로 설정한다.(단계5 참고)

b. pages마당의 값을 nr_swap_pages와 total_swap_pages변수에 더한다.(단계6 참고)

c. path_release()를 호출하여 단계 2의 path_walk()에서 할당한 VFS객체를 해제한다.

d. 끝으로 오유코드를 반환한다.

9. 그렇지 않으면 사용중인 모든 페지슬로트를 RAM으로 옮기는 일을 성공한것이다. 따라서 함수는 다음과 같은 단계를 실행한다.

a. specialfile이 블로크장치파일을 나타내면 대응하는 블로크장치구동프로그람을 해제한다.

b. path_release()를 호출하여 단계 2의 path_walk()에서 할당한 VFS객체를 해제한다.

c. swap_map 배렬을 저장하는데 사용한 기억기령역을 해제한다.

d. path_release()를 다시 호출한다. specialfile을 가리키는 VFS 객체를 sys_swapon()이 호출한 path_walk()함수에서 할당했을수 있기때문이다.

e. 0을 반환한다.(성공)

◦ try_to_unuse()함수

앞에서 살펴본것처럼 try_to_unuse()함수는 페지를 교환하여 넣기하고 교환하여

내보낸 모든 페지의 표를 갱신한다.

함수는 모든 핵심부스레드와 프로쎄스의 주소공간을 방문하며 표시로 사용하는 init_mm기억기서술자부터 시작한다. 이 함수는 새치기가 활성화된 상태에서 오래동안 실행되는 함수이다. 따라서 다른 프로쎄스와의 동기화가 중요하다. try_to_unuse()함수는 교환령역의 swap_map 배렬을 순차적으로 탐색한다. 함수가 시용중인 페지슬로트를 발견하면 페지를 교환하여 넣기하고 이 페지를 참조하는 프로쎄스를 찾기 시작한다. 경쟁조건(race condition)을 피하기 위해 이 두 연산의 순서를 지켜야 한다. 입출력자료전송중에는 페지에 잠그기가 걸리므로 다른 프로쎄스는 페지에 접근할수 없다. 입출력자료전송이 완료되면 try_to_unuse()가 페지를 다시 잠그기때문에 다른 핵심부조종경로에 의해 교환하여 내보낼수 없다. 또한 각 프로쎄스가 교환하여넣기나 교환하여 내보내기 연산을 시작하기 전에 페지캐쉬를 탐색하므로 경쟁조건을 피할수 있다.(《교환캐쉬》참고) 그리고 try_to_unuse()가 접근하는 교환령역은 쓰기금지로 표시해두기때문에(SWP_WRITEOK기발을 설정하지 않음) 다른 프로쎄스가 이 령역의 페지슬로트를 교환하여 내보낼수 없다.

그러나 try_to_unuse()는 교환령역의 사용계수기배렬 swap_map을 여러번 탐색해야 할수도 있다. 교환하여 내보낸 페지참조를 포함하는 기억기령역이 프로쎄스목록에서 한번 탐색할 때에는 나타나지 않았다가 다음에 탐색할 때에는 다시 나타날수 있다.

례를 들어 do_munmap()함수의 설명을 다시 보자.(《선형주소구간해제》참고) 프로쎄스가 선형주소구간을 해제하면 do_munmap()은 해제대상인 선형주소를 포함하는 모든 기억기령역을 프로쎄스목록에서 제거한다. 그리고 함수는 부분적으로 배치가 해제된 기억기령역을 다시 프로쎄스목록에 삽입한다. do_munmap()은 해제된 선형주소구간에 속하는 교환하여 내보낸 페지의 해제를 처리한다. 그러나 다행히 프로쎄스목록에 다시 삽입해야 하는 기억기령역에 속하는 교환하여 내보낸 페지는 해제하지 않는다.

따라서 try_to_unuse()는 대응하는 기억기령역이 림시로 프로쎄스목록에서 빠져있어서 주어진 페지슬로트를 참조하는 프로쎄스를 찾는데 실패할수 있다. 이 상태를 해결하기 위해 try_to_unuse()는 모든 참조계수기가 0이 될 때까지 swap_map 배렬의 탐색을 계속한다.

결국 프로쎄스목록에서 사라진 교환하여 내보낸 페지를 참조하는 기억기령역이 다시 나타나게 되고 try_to_unuse()는 모든 페지슬로트를 해제하는데 성공한다.

이제 try_to_unuse()가 실행하는 주요 연산을 보자. 이 함수는 변수로 받은 교환령역의 swap_map 배렬에 있는 참조계수기에 대해 순환을 계속 반복한다. 각 참조계수기에 대해 다음 단계를 수행한다.

1. 계수기가 0이거나(페지가 저장되여있지않음) SWAP_MAP_BAD면 다음 페지슬로트에 대해 계속한다.

2. 그렇지 않으면 read_swap_cache_async()함수를 호출하여(뒤에 나오는《교환

페지전송》참고) 페지를 교환하여 넣기한다. 이 과정에서 새로운 페지틀이 필요하면 할당하고 페지틀을 페지슬로트에 저장되여있던 자료로 채우고 페지를 교환캐쉬에 넣는다.

3. 새로운 페지가 디스크의 내용으로 바뀔 때까지 기다린 다음 페지에 잠그기를 건다.

4. 앞의 단계를 실행하는중에 프로쎄스가 보류될수 있으므로 참조계수기가 0인지 다시 검사한다. 0이면 다음 페지슬로트에 대해 계속한다.(다른 핵심부조종경로에서 이미 이 교환페지를 해제하였다.)

5. init_mm이 나타내는 2중련결목록의 모든 기억기서술자에 대해 unuse_process()를 호출한다.(《기억기서술자》참고) 이 함수는 많은 시간을 소비하며 기억기서술자를 소유하는 프로쎄스의 모든 페지표입구점을 탐색하여 교환하여 내보낸 페지식별자를 모두 페지틀의 물리적주소로 변경한다. 함수는 이 변경을 반영하기 위해 swap_map배렬의 페지슬로트 계수기를 감소시키고 (SWAP_MAP_MAX인 경우는 례외) 페지틀의 사용계수기를 증가시킨다.

6. shmem_unuse()를 호출하여 교환하여 내보낼 페지가 IPC공유기억기자원으로 사용되는지 검사하고 그 경우에 대해 적절히 처리하게 한다.(《IPC 공유기억기》 참고)

7. 페지의 참조계수기의 값을 검사한다. 값이 SWAP_MAP_MAX면 페지슬로트는 영구적이다. 이 경우를 해제하기 위해 참조계수기를 1로 만든다.

8. (참조계수기의 값에 따라) 교환캐쉬도 페지를 포함하고있을수 있다. 페지가 교환캐쉬에 속하면 rw_swap_page()함수를 호출하여 내용을 디스크에 흘리도록 하고(페지가 불결한 경우) delete_from_swap_cache()를 호출하여 페지를 교환캐쉬에서 제거하도록 하고 사용계수기를 감소시킨다.

9. 페지서술자의 PG_dirty기발을 설정하고 페지의 잠그기를 해제한다.

10. 현재프로쎄스의 need_resched마당을 검사한다. 마당이 설정되여있으면 schedule()을 호출하여 CPU를 풀어주도록 한다. 교환령역을 비활성화하는것은 오래 걸리는 작업이므로 핵심부는 체계의 다른 프로쎄스가 계속 실행되도록 해야 한다. 순서짜기프로그람이 현재프로쎄스를 다시 선택하면 try_to_unuse()함수는 이 단계에서 계속 실행한다.

11. 계속하여 다음 페지슬로트에 대해서 1단계에서 시작한다.

이 함수는 swap_map배렬의 모든 참조계수기가 0이 될 때까지 계속한다. 다음페지슬로트에 대한 검사를 시작하였다 하더라도 이전페지슬로트의 참조계수기가 여전히 정수일수 있다. 5단계에서 일부기억기령역이 탐색한 프로쎄스목록에서 림시로 제거될수 있으므로 유령(ghost)프로쎄스가 페지를 계속 참조할수 있다. 결국 try_to_unuse()는 모든 참조를 처리한다. 그러나 도중에는 더는 교환캐쉬에 없으며 잠그기가 풀려있고 비활성화되고있는 교환령역의 페지슬로트에 복사본을 포함한 페지가 존재한다.

이 상태가 결국 자료손실을 가져온다고 생각할수도 있다. 례를 들어 어떤 유령프로쎄스가 페지슬로트에 접근하여 페지의 교환하여 넣기를 시작하였다고 가정하자. 페지가

교환캐쉬에 없으므로 프로쎄스는 새로운 페지틀을 디스크에서 읽은 자료로 채운다. 그러나 이 페지틀은 유령프로쎄스와 페지를 공유하는것으로 여겨지는 프로쎄스가 소유한 페지틀과 다를수 있다.

이 문제는 교환령역을 비활성화하는동안에 발생하지 않는다. 유령 프로쎄스로부터의 간섭은 교환하여 내보내는 페지가 비공개닉명기억기배치에 속하는 경우에만 발생하기때문이다. 이 경우 페지틀은 《쓰기복사》기법에 의해 처리되므로 페지를 참조하는 프로쎄스에 다른 페지틀을 할당하는것은 전혀 문제가 되지 않는다. 그러나 try_to_unuse()함수는 페지를 불결한것으로 표시한다.(9단계)

그렇지 않으면 try_to_swap_out()함수가 후에 어떤 프로쎄스의 페지표에서 페지를 다른 교환령역에 저장하지 않고 삭제해 버릴수있다.(《페지를 교환하여 내보내기》참고)

4) 페지슬로트의 할당과 해제

뒤에서 보지만 기억기를 비울 때 핵심부는 짧은 시간동안에 많은 페지를 교환하여 내보낸다. 따라서 교환령역에 접근할 때 디스크탐색시간을 최소화하기 위해 이 페지들을 련속하는 슬로트에 저장하도록 시도하는것이 중요하다.

여유슬로트를 찾는 알고리듬의 접근방법으로 다음과 같이 간단한 두가지 전략중 하나를 선택할수 있다.

- 언제나 교환령역의 처음부터 시작한다. 여유페지슬로트가 서로 멀리 떨어져있을수 있으므로 이 접근방법은 교환하여 내보내기연산중의 평균탐색시간을 증가시킬수 있다.
- 언제나 이전에 마지막으로 할당된 페지슬로트부터 시작한다. 이 접근방법은 대부분의 교환령역이 여유있다면(대부분의 경우이다.) 채워져있는 몇 안되는 페지슬로트가 서로 멀리 떨어져있을수 있으므로 교환하여넣기연산도중의 평균 탐색시간을 증가시킬것이다. Linux는 혼합된 접근방법을 사용한다. Linux는 언제나 이전에 마지막으로 할당된 페지슬로트에서 시작하지만 다음의 경우는 례외이다.

- 교환령역의 끝에 도달했을 때
- 가장 최근에 다시 교환령역의 시작위치에서 할당을 시작한 후 여유페지슬로트를 SWAPFILE_CLUSTER(보통256)개 할당했을 때

swap_info_struct서술자의 cluster_nr마당은 할당된 여유페지슬로트의 수를 저장한다. 함수가 다시 교환령역의 시작위치에서 할당을 시작하면 이 마당을 0으로 초기화한다. cluster_next마당은 다음 할당에서 검사할 첫번째 페지슬로트의 색인값을 저장한다.

핵심부는 여유페지슬로트를 찾는 속도를 높이려고 각 교환령역서술자의 lowest_bit와 highest_bit마당을 최신상태로 유지한다. 이 마당은 여유있을수 있는 처음과 마지막 페지슬로트를 나타낸다. 다시 말해서 1owest_bit아래와 highest_bit우에 있는 모든 페지슬로트는 할당된 상태이다.

○ **scan_swap_map()함수**

어떤 교환령역에서 여유페지슬로트를 찾기 위해 scan_swap_map()을 사용한다.

이 함수는 교환령역서술자를 가리키는 변수를 받아서 여유페지슬로트의 색인값을 반환한다. 교환령역에 여유슬로트가 없으면 0을 반환한다. 함수는 다음과 같은 단계를 실행한다.

1. 현재클라스터를 사용하려고 시도한다. 교환령역서술자의 cluster_nr마당이 정수이면 계수기배렬인 swap_map에서 cluster_next색인값이 가리키는 요소부터 시작하여 빈 입구점을 찾는다. 빈 입구점을 찾으면 cluster_nr마당을 감소시키고 4단계로 간다.

2. 이 단계에 이르면 cluster_nr 마당이 0이거나 cluster_next에서 시작한 탐색이 swap_map배렬에서 빈 입구점을 찾지 못한 경우이다. 이제 혼합탐색의 두번째 단계를 시도할 차례이다. cluster_nr을 SWAPFILE_CLUSTER로 다시 초기화하고 1owest_bit색인값에서부터 여유페지슬로트 SWAPFILE_CLUSTER개로 구성된 그룹을 찾아본다. 이러한 그룹을 찾으면 4단계로 간다.

3. 여유페지슬로트 SWAPFILE_CLUSTER개로 이루어진 그룹을 찾지 못하였다. 여유페지슬로트 하나를 찾기위해 1owest_bit색인값에서부터 배렬을 다시 찾기 시작한다. 빈 입구점을 찾을수 없으면 1owest_bit마당을 배렬의 최대 색인값으로 highest_bit마당을 0으로 설정하고 0을 반환한다.(교환령역이 모두 찼다.)

4. 빈 입구점을 찾았다. 입구점안에 1을 넣고 nr_swap_pages를 감소시킨다. 1owest_bit와 highest_bit마당을 적절히 변경하고 cluster_next마당을 방금 할당한 페지슬로트의 색인값+1로 설정한다.

5. 할당한 페지슬로트의 색인값을 반환한다.

○ get_swap_page()함수

get_swap_page()함수는 모든 활성교환령역을 검색하여 여유페지슬로트 하나를 찾아준다. 이 함수는 새로 할당한 페지슬로트의 색인값을 반환해주거나 모든 교환령역이 꽉 차있으면 0을 반환한다. 함수는 활성화상태인 교환령역들의 서로 다른 우선순위를 고려한다.

이 함수는 두번의 과정(pass)으로 이루어진다. 첫번째 과정은 부분적이며 같은 우선순위의 령역에만 적용된다. 함수는 그런 령역을 원형방식으로 돌아가면서 여유슬로트를 찾는다. 여유페지슬로트를 찾을수 없으면 두번째 과정을 교환령역목록의 처음부터 시작한다. 두번째 과정에서는 모든 교환령역을 검사한다. 이 함수는 다음과 같은 단계를 실행한다.

1. nr_swap_pages가 0이거나 활성교환령역이 없으면 0을 반환한다.

2. swap_list.next가 가리키는 교환령역부터 검토하기 시작한다.(교환령역은 높은 우선순위부터 우선순위가 감소하는 방향으로 정렬되여있다.)

3. 교환령역이 활성회된 상태고 비활성화하는 도중이 아니면 scan_swap_map()을 호출하여 여유페지슬로트를 할당한다. scan_swap_map()이 페지슬로트색인값를 반환하면 함수의 작업은 사실상 완료되였다. 그러나 다음 호출에 준비해야 한다. 다음교환령역

의 우선순위가 현재교환령역과 같으면 swap_list.next가 교환령역목록의 다음교환령역을 가리키도록 변경한다.(이렇게 해서 이 교환령역들을 원형방식으로 사용할수있다.) 다음교환령역이 현재교환령역과 우선순위가 같지 않다면 함수는 목록의 첫번째 교환령역으로 swap_list.next를 설정한다.(다음 탐색은 우선순위가 가장 높은 교환령역에서 시작한다). 함수는 방금 할당한 페지슬로트에 대응하는 식별자를 반환하면서 완료한다.

4. 여기에 이르면 교환령역에 쓰기금지가 되여있거나 여유페지슬로트가 없다. 교환령역목록의 다음교환령역이 현재교환령역과 우선순위가 같다면 다음교환령역을 현재교환령역으로 설정하고 3단계로 이동한다.

5. 여기에 이르면 교환령역목록의 다음교환령역은 이전것보다 우선순위가 낮다. 다음 단계는 함수가 두 과정중 어느 과정을 실행하는가에 따라 달라진다.

a. 현재 첫번째(부분적) 과정라면 목록의 첫번째 교환령역을 대상으로 한다. 3단계로 이동해서 두번째 과정을 시작한다.

b. 그렇지 않으면 두번째 과정이다. 목록에 다음 요소가 있는지 검사한다. 다음 요소가 있다면 이것을 대상으로 3단계로 이동한다.

6. 여기에 이르면 두번째 과정으로 목록을 모두 탐색했고 여유페지슬로트를 발견하지 못했으므로 0을 반환한다.

swap_free()함수는 페지를 교환하여넣기할 때 대응하는swap_map계수기를 감소시키기 위해 사용한다.(표 4-12 참고) 계수기가 0이 되면 페지슬로트 식별자가 더는 어떤 페지표입구점에도 포함되지 않으므로 페지슬로트는 여유있는것이다. 그러나 교환캐쉬에서 보지만 교환캐쉬도 페지슬로트의 소유자로 취급한다.

이 함수는 교환하여 내보낸 페지식별자를 나타내는 entry변수를 받으며 다음과 같은 단계를 실행한다.

1. entry에서 교환령역색인값과 편위 페지슬로트색인값을 얻고 교환령역서술자의 주소를 얻는다.

2. 교환령역이 활성화상태인지 검사한다. 그렇지 않으면 즉시 되돌이한다.

3. 해제중인 페지슬로트에 대응하는 swap_map계수기가 SWAP_MAP_MAX보다 작으면 계수기를 감소시킨다. SWAP_MAP_MAX값을 담은 입구점은 영구적이라고 취급한다.(삭제할수 없다.)

4. swap_map계수기가 0이되면 nr_swap_pages를 증가시키고 필요하다면 교환령역서술자의 lowest_bit와 highest_bit마당을 변경한다.

3. 교환캐쉬

교환캐쉬는 교환중인 페지에 접근하려는 프로쎄스들사이의 경쟁조건(race condition)을 피하기 위해서 꼭 필요하다.

어떤 페지를 한 프로쎄스가 소유하고있으면(또는 해당 페지가 하나이상의 복제프로

쎄스가 소유한 주소공간에 속해있으면) 고려해야 하는 경쟁조건은 하나뿐이다. 즉 프로쎄스가 교환하여 내보내기중인 페지에 접근하려고 하는 경우이다. 각 페지슬로트마다 신호기 하나가 대응하는 신호기배렬을 사용하여 입출력자료 전송이 끝날 때까지 프로쎄스를 차단할수 있다.

그러나 한 페지를 여러 프로쎄스가 소유하는 경우가 많다. 핵심부가 교환하여 내보내기중인 페지를 참조하는 모든 페지표입구점을 빨리 찾을수 있으면 우의 신호기 배렬만으로 경쟁조건을 피할수 있다. 이렇게 함으로써 핵심부는 모든 프로쎄스가 같은 페지틀과 교환하여 내보낸 페지식별자를 볼수 있게 한다.

Linux 2.6에는 어떤 페지틀에 대해 이 페지틀을 소유한 모든 프로쎄스의 목록을 얻을수 있는 빠른 방법이 없다. 모든 프로쎄스의 모든 페지표입구점을 탐색하여 주어진 물리적주소를 가진 항목을 찾는것은 아주 높은 비용이 들기때문에 아주 가끔 수행한다.(례를 들면 교환령역을 비활성화할 때)

결국 같은 페지가 어떤 프로쎄스에 대해서는 교환하여 내보내여있고 어떤 프로쎄스에 대해서는 기억기에 존재하는 상태가 발생한다. 핵심부는 이렇게 이상한 상태에서 발생하는 경쟁조건을 피하려고 교환캐쉬를 사용한다. 교환캐쉬가 어떻게 동작하는지 설명하기 전에 어떤 경우에 여러 프로쎄스가 페지틀을 공유하는지 다시 보자.

・페지틀이 공유실명(shared nonanonymous)기억기배치와 관련된 경우(《기억기배치》참고)

・새로운 프로쎄스를 생성(fork)한 경우 또는 페지틀이 비공개기억기배치에 속한 경우처럼 《쓰기복사》로 페지틀을 처리하는 경우(《쓰기복사》참고)

・페지틀을 IPC공유기억기자원에 할당한 경우(《IPC공유기억기》참고) 또는 공유닉명기억기배치에 관련된 경우

물론 여러 프로쎄스가 기억기서술자를 공유하고 따라서 전체 페지표를 공유하는 경우 페지틀도 공유하게 된다. CLONE_VM기발을 전달하여 clone()체계호출을 실행하면 이런 프로쎄스들이 생성된다.(《clone(), fork(), vfork()체계호출》 참고) 그러나 교환알고리듬은 모든 복제프로쎄스를 한 프로쎄스로 취급한다. 따라서 여기서 프로쎄스들을 말할 때에는 서로 다른 기억기서술자를 소유하고있는 프로쎄스들을 의미한다.

뒤에서 보지만 공유실명기억기배치에 사용하는 페지틀은 교환하여 내보내지 않는다. 대신 다른 핵심부함수가 페지틀자료를 적절한 파일에 저장하고 페지틀을 제거한다. 그러나 다른 두 종류의 공유페지틀은 교환알고리듬에서 교환캐쉬를 사용하여 주의깊게 처리해야 한다.

교환캐쉬는 교환령역에 복사된 공유페지틀들을 의미한다. 별도의 자료구조로 존재하지 않고 대신 정규페지캐쉬에 있는 페지의 특정한 마당이 설정되여있으면 해당 페지가 교환캐쉬에 들어있는것으로 취급한다.

공유된 페지의 교환은 다음과 같이 동작한다. 두 프로쎄스 A와 B가 어떤 페지 P를

공유한다고 하자. 교환알고리듬이 프로쎄스 A의 페지틀을 탐색하여 페지 P를 교환하여 내보낼 대상으로 선택하였다면 알고리듬은 새로운 페지슬로트를 할당하고 P에 저장된 자료를 새로운 페지슬로트로 복사한다. 그리고 교환하여 내보낸 페지식별자를 프로쎄스 A의 대응하는 페지표입구점에 저장한다. 끝으로 __free_page()를 호출하여 페지틀을 해제한다. 그러나 프로쎄스 B도 페지 P를 소유하고있기때문에 페지의 사용계수기는 0이 되지 않는다. 따라서 교환알고리듬은 교환령역으로 페지를 전송하는데는 성공했지만 대응하는 페지틀을 회수하는데는 실패한다.

후에 교환알고리듬이 프로쎄스 B의 페지틀을 탐색하여 페지 P를 교환하여 내보내기로 선택하였다고 가정하자. 페지를 다시 교환하여 내보내지 않으려면 핵심부가 페지 P가 이미 교환령역으로 전송되였다는 사실을 알아야 한다. 또한 페지슬로트의 사용계수기를 증가시키기 위해서는 교환하여 내보낸 페지식별자를 얻을수 있어야 한다.

그림 4-23은 서로 다른 시간에 여러 프로쎄스에서 교환하여내보낸 어떤 공유페지에 대해 핵심부가 수행하는 작업을 보여준다. 교환령역과 페지 P안의 수자는 각각 페지슬로트의 사용계수기와 페지의 사용계수기를 나타낸다. 각 사용계수기는 페지나 페지슬로트를 사용하는 모든 프로쎄스의 수를 포함하고 페지가 교환캐쉬에 포함되여있으면 교환캐쉬도 포함한다. 그림에서 4단계를 볼수있다.

1. (a)에서 P는 A와 B의 페지표에 모두 존재한다.

2. (b)에서 P는 A의 주소공간에서 교환하여 내보냈다.

3. (c)에서 P는 A와 B의 주소공간에서 모두 교환하여 내보냈지만 여전히 교환캐쉬에 있다.

4. 마지막으로 (d)에서 P는 해제되여 형제체계로 돌아간다.

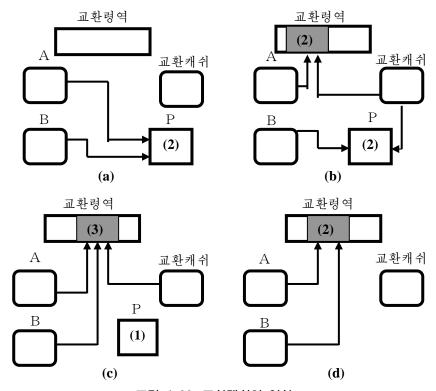



그림 4-23. 교환캐쉬의 역할

교환캐쉬는 《페지캐쉬》에서 설명한 페지캐쉬자료구조와 처리부를 실현한다. 다시 말하지만 페지캐쉬의 핵심은 페지의 소유자를 식별하는 address_space객체의 주소와 편위값으로부터 알고리듬이 페지서술자의 주소를 빨리 얻도록 하는 하쉬표이다.

교환캐쉬에 있는 페지는 페지캐쉬의 다른 페지와 마찬가지로 저장되지만 다음과 같이 특별하게 처리한다.

·페지서술자의 mapping마당은 swapper_space변수에 저장된 address_space객체를 가리킨다.

·index마당은 페지에 대응하는 교환하여 내보낸 페지식별자를 저장한다.

또한 페지가 교환캐쉬에 들어가게 되면 교환캐쉬가 페지틀과 페지슬로트 둘다 사용하므로 페지서술자의 count마당과 페지슬로트사용계수기를 증가시킨다.

1) **교환캐쉬보조함수**

핵심부는 교환캐쉬를 처리하기 위해 몇가지 함수를 사용한다. 이 함수들은 주로 앞의 《페지캐쉬》에서 설명한 내용을 기반으로 한다. 뒤에서 고수준함수가 페지를 교환하여 넣기/내보내기위해 필요에 따라 저수준함수를 어떻게 호출하는지 살펴본다.

교환캐쉬를 처리하는 주요함수는 다음과 같다.

lookup_swap_cache()

변수로 전달된 교환하여 내보낸 페지식별자를 교환캐쉬에서 찾아서 페지주소를 반환한다. 페지가 캐쉬에 없으면 0을 반환한다. 요청한 페지를 찾기 위해 swapper_space 페지주소공간객체의 주소와 교환하여 내보낸 페지식별자를 변수로 find_get_page()를 호출한다.

add_to_swap_cache()

페지교환캐쉬에 넣는다. swap_duplicate()를 호출하여 변수로 받은 페지슬로트가 유효한지 검사하고 페지슬로트사용계수기를 증가시킨다. find_get_page()를 호출하여 address_space객체와 편위가 같은 페지가 이미 캐쉬에 있는지 확인한다. add_to_swap_cache()를 호출하여 페지를 캐쉬에 넣는다. lru_cache_add()를 호출하여 페지를 비활성목록에 넣는다.(《LRU목록》 참고)

delete_from_swap_cache()

내용을 디스크에 흘리고 PG_dirty기발을 해제하고 remove_page_from_inode_queue()와 remove_page_from_hash_queue()를 호출하여 페지를 교환캐쉬에서 제거한다.(《페지캐쉬자료구조》참고)

free_page_and-swap_cache()

__free_page()를 호출하여 페지를 해제한다. 호출한 프로쎄스가 페지를 소유한 유일한 프로쎄스이면 함수는 페지를 활성 또는 비활성목록에서 제거하고 (《LRU목록》참고) delete_from_swap()를 호출하여 교환캐쉬에서 페지를 제거하고 페지내용을 디스크로 흘리고 swap_free()를 호출하여 교환령역의 페지슬로트를 해제한다.

4. 교환페지 전송

교환페지의 전송은 아주 많은 경쟁조건(race condition)과 기타 방지해야 할 잠재적인 위험(hazard)때문에 아주 복잡하다. 주기적으로 다음과 같은 사항을 검사해야 한다.

・어떤 페지에 대한 교환하여 넣기 또는 교환하여 내보내기도중 페지를 소유하고있는 프로쎄스가 완료할수 있다.

・어떤 프로쎄스가 교환하여 내보내기를 시도하는중에 다른 프로쎄스가 해당 페지교환하여 넣거나 반대로 어떤 프로쎄스가 교환하여 넣기를 시도하는중에 다른 프로쎄스가 해당 페지를 교환하여 내보내려 할수있다.

다른 디스크접근류형과 마찬가지로 교환페지에 대한 입출력자료전송도 차단연산이다. 따라서 핵심부는 같은 페지들, 같은 페지슬로트 또는 둘 모두가 관련된 동시적인 전송을 피해야 한다.

앞에서 설명한 기구를 통해 페지들에 대한 경쟁조건을 피할수 있다. 특히 페지들에 대한 입출력연산을 시작하기 전에 핵심부는 PG_locked 기발이 해제될 때까지 기다린다. 함수가 완료하는 과정에서 페지들잠그기를 얻으므로 다른 핵심부조종경로는 입출력연산동안 이 페지들의 내용에 접근할수 없다.

그러나 페지슬로트의 상태를 추적할수 있어야 한다. 입출력자료전송과 관련된 페지슬로트에 대한 배타적인 접근을 보장하기 위해 페지서술자의 PG_locked 기발을 다시 사용한다. 교환페지에 대해 입출력연산을 시작하기 전에 핵심부는 관련된 페지틀이 교환캐쉬에 포함되여있는지 검사한다. 포함되여있지 않으면 핵심부는 페지틀을 교환캐쉬에 추가한다. 어떤 페지를 전송중인데 어떤 프로쎄스가 해당 페지를 교환하여 넣기하려 한다고 가정하자. 교환하여넣기와 관련한 작업을 시작하기 전에 핵심부는 주어진 교환하여 내보낸 페지식별자에 대응하는 페지틀을 교환캐쉬에서 찾는다. 페지틀을 찾는 경우 핵심부는 새로운 페지틀을 할당하지 않고 캐쉬된 페지틀을 사용해야 한다. 그리고 PG_locked 기발이 1로 설정되여있기때문에 핵심부는 이 기발비트가 0이 될 때까지 핵심부 조종경로를 보류하여 입출력연산을 완료할 때까지 페지틀의 내용과 교환령역에 있는 페지슬로트가 유지되도록 한다.

요약하면 교환캐쉬덕분에 페지틀의 PG_locked 기발은 교환령역의 페지슬로트에 대한 잠그기의 역할도 수행한다.

1) rw_swap_page()함수

페지를 교환하여 넣기 또는 교환하여 내보내기하는데는 rw_swap_page()함수를 사용한다. 이 함수는 다음과 같은 변수를 받는다.

rw

자료전송방향을 나타내는 기발. 교환하여 넣는 경우 READ, 교환하여 내보낼 경우 WRITE이다.

page

교환캐쉬에 있는 페지의 서술자주소.

함수를 호출하기 전에 호출하는 쪽에서는 해당 페지가 교환캐쉬에 포함되여있도록 해야하고 앞에서 설명한것처럼 페지틀이나 교환령역에 있는 페지슬로트에 대한 동시접근으로 인한 경쟁조건을 방지하기 위해 페지에 잠그기를 걸어야 한다. 안전하게 처리하기 위해 rw_swap_page()함수는 이 두 조건을 실제 만족하는지 검사하고 page->index에서 교환하여 내보낸 페지식별자를 얻고 페지식별자의 페지서술자주소페지방향 기발 fW를 변수로 rw_swap_page_base()함수를 호출한다.

rw_swap_page_base()함수는 교환알고리듬의 핵심으로 다음과 같은 단계를 수행한다.

1. 자료전송이 교환하여 넣기연산인 경우(rw가 RFAD이면) 페지틀의 PG_uptodate 기발을 해제한다. 교환하여 넣기연산을 정상적으로 완료할 때에만 이 기발을 다시 설정한다.

2. 교환하여 내보낸 페지식별자로부터 교환령역서술자와 슬로트색인값을 얻는다.

3. 교환령역이 디스크구획이면 교환령역서술자의 swap_device 마당에서 대응하는 블로크장치번호를 얻는다. 교환디스크구획의 블로크크기는 언제나 페지크기

(PAGE_SIZE)와 같으므로 이 경우 슬로트색인값은 요청한 자료의 론리적블로크번호을 나타낸다.

4. 그렇지 않으면 교환령역은 정규파일이다. 다음 단계를 수행한다.

a. 파일을 저장하고있는 블로크장치의 번호를 파일의 i마디객체의 i_dev 마당에서 얻는다.(교환령역서술자의 swap_files->d_inode 마당)

b. 장치의 블로크크기를 얻는다.(i마디의 i_sb->s_blocksize마당)

c. 주어진 슬로트색인값에 대응하는 파일블로크번호를 계산한다.

d. 페지슬로트에 있는 블로크들의 론리적블로크번호들로 지역배렬을 채운다. 매 론리적블로크번호는 i마디의 i_mapping마당에 저장된 주소가 가리키는 address_space객체의 bmap메쏘드를 호출하여 얻는다. bmap메쏘드가 실패하면 rw_swap_page_base()는 0을 반환한다.(실패)

5. brw_page()함수가 활성화하는 페지입출력연산이 비동기적이기때문에 rw_swap_page()함수는 실제 입출력자료전송이 완료되기전에 완료할수도 있다. 그렇지만 앞의 《페지입출력연산》에서 설명한것처럼 핵심부는 end_buffer_io_async()함수를 실행하고(모든 자료전송이 성공적으로 완료되였음을 확인한다.) 페지의 잠그기를 풀고 PG_uptodate기발을 설정한다.

2) read_swap_cache_async()함수

read_swap_cache_async()함수는 변수로 교환하여 내보낸 페지식별자를 받는다. 이 함수는 핵심부가 페지를 교환하여 넣기해야할 때 호출한다. 교환구획에 접근하기 전에 함수는 교환캐쉬가 원하는 페지들을 포함하고있는지 검사해야 한다. 이 함수는 다음과 같은 연산을 수행한다.

1. find_get_page()를 호출하여 페지를 교환캐쉬에서 찾는다. 페지를 찾았으면 페지서술자의 주소를 반환한다.

2. 페지가 교환캐쉬에 포함되여있지 않다. alloc_page()를 호출하여 새로운 페지틀을 할당한다. 여유페지틀이 없으면 0을 반환한다.(체계에 여유 기억기가 없음을 나타낸다.)

3. add_to_swap_cache()를 호출하여 페지틀을 교환캐쉬에 삽입한다. 앞서 《교환캐쉬보조함수》에서 설명한것처럼 이 함수는 페지에 잠그기를 건다.

4. 앞단계의 add_to_swap_cache()는 페지사본이 교환캐쉬에 있으면 실패할수 있다. 례를 들어 프로쎄스가 2단계에서 차단될수 있으며 다른 프로쎄스가 같은 페지슬로트에 대해 교환넣기연산을 시작할수 있다. 이 경우에 함수는 3단계에서 할당한 페지틀을 해제하고 1단계에서 다시 시작한다.

5. 그렇지 않으면 새로운 페지틀이 교환캐쉬에 삽입되였다. READ와 페지서술자를 변수로 전달해서 rw_swap_page()를 호출하여 페지의 내용을 교환 령역에서 읽게 한다.

6. 페지서술자의 주소를 반환한다.

3) rw_swap_page_nolock()함수

핵심부가 교환령역에서 페지를 읽기만 하고 그 내용을 교환캐쉬에 넣지 않으려는 경우가 있다. swapon()체계호출을 봉사하는 경우 이런 일이 발생한다. 핵심부는 swap_header를 포함하는 교환령역의 첫번째 페지를 읽고 즉시 해당 페지틀을 버린다. 핵심부가 교환령역을 활성화하는 중이기때문에 다른 프로쎄스가 페지를 교환령역에 교환하여 넣기 또는 교환하여 내보낼수 없다. 따라서 페지슬로트에 대한 접근을 막을 필요가 없다. rw_swap_page_nocache()함수는 변수로 입출력연산의 류형(READ 또는 WRITE), 교환하여 내보낸 페지식별자, 페지틀주소(이미 잠그기가 걸린)를 포함한다. 이 함수는 다음 연산을 실행한다.

1. 변수로 전달받은 페지틀의 페지서술자를 얻는다.
2. 페지서술자의 swapping 마당을 swapper_space객체의 주소로 설정한다. 4단계에서 sync_page메쏘드를 실행하기때문이다.
3. 입출력교환연산을 시작하기 위해 rw_swap_page_base()를 호출한다.
4. wait_on_page()를 호출하여 입출력자료전송이 완료될 때까지 기다린다.
5. 페지의 잠그기를 해제한다.
6. 페지서술자의 mapping마당을 NULL로 설정하고 되돌이한다.

5. 페지를 교환하여 내보내기

뒤에 나오는《페지틀비우기》에서 페지가 교환하여 내보내는 과정에서 어떤 일을 하는지 설명한다. 이절의 시작부분에서 지적한것처럼 페지를 교환하여 내보내는것은 최후의 수단이며 기억기를 제거하는 여러가지 일반적인 기법중 하나이다. 여기서는 핵심부가 교환하여 내보내기를 실행하는 방법을 살펴본다. 교환하여 내보내기는 여러 함수를 련속적으로 호출하여 수행된다. 그럼 높은 수준의 함수부터 시작하자.

swap_out()함수는 어떤 기억기지역(zone)에서 페지를 교환하여 내보낼것인지 나타내는 classzone변수를 받는다. priority와 gft_mask변수는 사용되지 않는다.

swap_out()함수는 현재 존재하는 기억기서술자를 탐색하여 각 프로쎄스의 페지표이 참조하는 페지를 교환하여 내보내려 한다. 함수는 다음 조건중 하나를 만족하면 완료한다.

・함수가 SWAP_CLUSTER_MAX(기본값은 32)만큼의 페지틀을 해제하는데 성공한다. 페지틀을 공유하던 모든 프로쎄스의 페지표에서 제거되면 해당 페지틀은 해제된것이다.

・함수가 기억기서술자를 n개 탐색하였다. 여기서 n은 함수가 시작할 때의 기억기서술자목록의 길이다.

모든 프로쎄스가 공평하게 swap_out()으로 인한 손실을 분담하도록 함수는 마지막 호출에서 참조한 기억기서술자부터 탐색을 시작한다. 이 기억기서술자의 주소를 대역변

수 swap_mm에 저장한다.

탐색하면서 검사하는 각 기억기서술자 mm에 대해 swap_out()함수는 mm->mm_users 사용계수기를 증가시켜서 교환알고리듬이 처리하는동안 기억기서술자가 목록에서 사라지는것을 방지한다. 그리고 swap_out()은 기억기서술자 주소 mm 기억기지역 classzone, 해제할 페지들의 수를 변수로 전달하여 swap_out_mm()함수를 호출한다. swap_out_mm()에서 복귀하면 swap_out()은 mm->mm_sers사용계수기를 감소시키고 목록의 다음기억기서술자를 분석할지 아니면 완료할지 결정한다.

swap_out_mm()은 기억기서술자를 소유한 프로쎄스의 함수가 해제한 페지의 수를 반환한다. swap_out()함수는 이 값을 함수실행시작부터 얼마나 많은 페지를 해제했는지 세는 계수기값을 변경하기 위해 사용한다. 계수기가 SWAP_CLUSUET_에 이르면 swap_out()은 완료한다.

swap_out_mm()함수는 변수로 전달받은 기억기서술자 mm을 소유한 프로쎄스의 기억기령역을 탐색한다. 보통 이 함수는 mm->mmap 목록의 첫번째 기억기령역 객체를 분석하기 시작한다.(이 목록은 시작선형주소순서로 정렬되여있다.) 그러나 mm이 이전 swap_out()호출에서 분석한 기억기령역이라면 swap_out_mm()은 첫번째 기억기령역부터 다시 시작하지 않고 이전의 호출에서 마지막으로 분석한 선형주소를 포함하는 기억기령역에서 시작한다. 이 선형주소는 기억기서술자의 swap_address 마당에 저장되여있다. 프로쎄스의 모든 기억기령역을 분석했으면 이 마당의 값은 TASK_SIZE가 된다.

swap_out_mm()은 기억기서술자 mm을 소유한 프로쎄스의 각 기억기령역에 대해서 아직 해제하지 않은 페지의 수, 분석할 첫번째 선형주소, 기억기령역객체, 기억기서술자를 변수로 swap_out_vma()함수를 호출한다. swap_out_vma()는 기억기령역에 속한 페지중 해제한 페지의 수를 반환한다. swap_out_mm()의 순환은 요청한 페지수만큼 페지를 해제했거나 모든 기억기령역을 검토할 때까지 계속된다.

swap_out_vma()함수는 기억기령역을 교환할수 있는지(례를 들면 VM_RESERVED가 0인지) 검사한다. 그리고 기억기령역에 있는 선형주소를 가리키는 프로쎄스의 페지대역등록부(Page Global Directory)의 모든 입구점을 검토하는 순차적인 단계를 시작한다. 각 입구점에 대해서 swap_out_pgd()함수를 호출하고 이 함수는 다시 기억기령역의 주소구간에 대응하는 페지중간등록부(Page Middle Directory)의 모든 입구점을 차례로 검토한다. swap_out_pgd()는 각 입구점에 대해 swap_out_pmd()함수를 호출하여 기억기령역의 페지를 참조하는 페지표에 담긴 모든 입구점을 검토한다. 또한 swap_out_pmd()는 try_to_swap_out()함수를 호출하여 마침내 페지에 대한 교환하여 내보내기를 시도한다. 보통 이 함수호출의 련속은 요청한 수만큼 페지틀을 해제하면 즉시 완료한다.

trv_to_swap_out()함수

try_to_swap_out()함수는 주어진 페지틀을 버리거나 그 내용을 교환하여 내보내기

함으로써 주어진 페지틀을 해제하려고 한다. 함수는 페지를 해제하는데 성공하면 1, 실패하면 0을 반환한다. 페지해제(release)란 페지를 공유한 모든 프로쎄스의 페지표에서 해당 페지틀에 대한 참조를 제거하는것을 의미한다. 이 경우 페지틀이 반드시 형제체계로 반환될 필요는 없다. 례를 들어 교환캐쉬가 계속 참조할수 있다.

이 함수의 변수는 다음과 같다.

mm

프로쎄스서술자주소

vma

기억기령역객체주소

address

페지의 제일처음의 선형주소

page_table

address를 배치한 페지표입구점주소

page

페지서술자주소

classzone

페지를 교환하여 내보내는 기억기지역

try_to_swap_out()함수는 페지표 입구점에 포함된 Accessed와 Dirty 기발을 사용한다. 정규페지화에서 언급한것처럼 Accessed 기발은 CPU의 페지화유니트에서 각 읽기쓰기접근에 대해 자동으로 설정하고 Dirty 기발은 각 쓰기접근에 대해 자동으로 설정한다. 이 두 기발은 기본적인 하드웨어지원을 제공하여 핵심부가 단순한 LRU 교체알고리듬을 실현할수있도록 한다.

try_to_swap_out()은 서로 다른 응답을 요구하는 서로 다른 상태을 반드시 인식해야 하지만 응답은 모두 같은 기본연산을 공유한다. 함수는 다음과 같은 단계를 수행한다.

1. page_table입구점의 Accessed기발을 검사한다. 기발이 1이면 페지를 젊은것(최근에 접근한것)으로 간주한다. 이 경우 Accessed 기발을 0으로 만들고 mark_page_accessed()를 호출하고(뒤에 나오는 《LRU목록》 참고) 0을 반환한다. 이 검사를 통해 페지는 이전 try_to_swap_out()호출이후로 페지에 접근한 사실이 없어야만 교환하여 내보낼수 있다.

2. 기억기령역에 잠그기가 걸려있으면(MV_LOCKED기발이 설정되여있음) 기억기령역에 대해 mark_page_accessed()를 호출하고 0을 반한한다.

3. page->flags마당의 PG_active기발이 설정되여있으면 페지는 자주 사용되고있으며 교환하여 내보내면 안된다. 함수는 0을 반환한다.

4. 페지가 classzone변수가 나타내는 기억기구역에 속하지 않으면 0을 반환한다.

5. 페지에 잠그기를 걸려고 시도한다. 이미 페지에 잠그기가 걸려있으면

(PG_1ocked 기발이 설정되여있음) 이 페지는 다른 입출력자료전송에 포함되여있으므로 페지를 교환하여 내보낼수 없다. 함수는 0을 반환한다.

6. 이제 함수는 페지를 교환하여 내보낼수 있다. page_table이 가리키는 페지표입구점을 0으로 설정하고 flush_tlb_page()를 호출하여 대응하는 TLB입구점을 무효화한다.

7. 페지표입구점의 불결한 기발이 설정되여있으면 set_page_dirty()함수를 호출하여 페지서술자의 PG_dirty기발을 설정한다. 그리고 이 함수는 page->mapping이 참조하는 addressss_space객체의 dirtys_pages목록에서 페지를 이동하고 i마디 page->mapping->host를 불결한것으로 표시한다.(《address_space 객체에 있는 페지서술자목록》 참고)

8. 페지가 교환캐쉬에 속해있으면 다음과 같이 실행한다.

a. page->index에서 교환하여 내보낸 페지의 식별자를 얻는다.

b. swap_duplicate()를 호출하여 페지슬로트색인값이 유효한지 검사하고 swap_map의 대응하는 사용계수기를 증가시킨다.

c. 교환하여 내보낸 페지식별자를page_table이 가리키는 페지표입구점에 기록한다.

d. 기억기서술자 mm의 rss마당을 감소시킨다.

e. 페지에 대한 잠그기를 해제한다.

f. 페지사용계수기 page->count를 감소시킨다.

g. 페지를 아무 프로쎄스에서도 참조하지 않으면 1을 반환한다. 그렇지 않으면 0을 반환한다. 다른 프로쎄스의 페지표를 탐색할 때 페지들이 이미 교환하여 내보내졌으므로 함수는 새로운 페지슬로트를 할당할 필요가 없다.

9. 페지가 교환캐쉬에 없다. 페지가 어떤 address_space객체에 속하는지 검사한다.(page->mapping마당이 NULL이 아니다.) 이 객체에 속한다면 페지는 공유파일기억기배치에 속해있으며 이 경우 함수는 8d 단계로 이동하여 페지틀을 해제하고 대응하는 페지표입구점을 NULL로 유지한다. 페지가 교환령역에 저장되지 않더라도 프로쎄스의 페지를참조를 해제할수 있다. 페지는 디스크에 영상을 가지고있으며 필요하다면 이 함수는 이미 7단계에서 영상에 대한 변경을 시작한다. 그리고 페지캐쉬가 계속 페지를 소유하고있으므로 페지틀은 형제체계로 반환되지 않는다.(《페지캐쉬관련 페지서술자마당》참고)

10. 함수가 여기에 이르면 페지는 교환캐쉬에 없고 addrcss_space 객체에 속하지도 않는다. 함수는 PG_dirty기발의 상태를 검사한다. 0이면 함수는 8d 단계로 이동하여 페지틀을 해제하고 대응하는 페지표입구점을 NULL로 유지한다. 프로쎄스가 페지틀에 쓰기를 한적이 없으므로 페지의 내용을 교환령역에 저장할 필요가 없다. PG_dirty 기발이 0이고 페지가 디스크에 영상을 가지지 않거나 페지가 비공개기억기배치에 속한 경우 이 기발은 초기화되지 않으므로 핵심부는 이 경우를 판단할수 있다. 프로쎄스가 같은 페지를 반복해서 접근하면 핵심부는 요구페지화기법을 사용하여 페지오유를 처리한

다.(《요구페지화》 참고) 새로운 페지들은 해제된 페지들에 저장된 자료로 채워진다.

11. 함수가 여기에 이르면 페지는 교환캐쉬에 없고 디스크에 영상을 가지지 않으며 불결하다. 함수는 페지가 완충기를 가지고있는지 검사한다.(페지는 완충기페지이며 page->buffers 마당이 NULL이 아니다.) 이 경우 함수는 페지표항목의 원래내용을 복구하고 페지의 잠그기를 해제하고 0을 반환한다. 페지가 address_space객체에 속하지 않고 디스크영상도 없으면서 어떻게 완충기를 가질수 있는가? 사실 이런 경우는 흔하지 않다. 례를 들면 방금 크기가 0으로 설정된(truncate) 파일의 부분을 배치하고있는 페지이다. 이 경우 try_to_swapout()은 아무 일도 하지 않는다.

12. 여기에 이르면 페지는 교환캐쉬에 없고 디스크에 영상을 가지지 않으며 불결하다. 함수는 반드시 이 페지를 새로운 페지슬로트에 교환하여 내보내야 한다. get_swap_page()를 호출하여 여유페지슬로트를 활성화된 교환령역중 하나에 할당한다. 여유페지슬로트가 없으면 페지표입구점의 원래 내용을 복구하고 페지에 대한 잠그기를 해제하고 0을 반환한다.

13. add_to_swap_cache()를 호출하여 페지를 교환캐쉬에 추가한다. 다른 핵심부 조종경로가 페지를 교환하여 넣기하려는 중이면 이 함수가 실패할수 있다. 다음에 보지만 다른 프로쎄스가 페지슬로트를 참조하지 않을 때도 실패하는 경우가 있다. 이 경우 swap_free()를 호출하여 페지슬로트를 해제하고 12단계에서 재시작한다.

14. 페지의 PG_uptodate기발을 설정한다.

15. add_to_swap_cache()가 PG_dirty기발을 초기화했으므로 set_page_dirty() 함수를 다시 호출한다.(앞에 있는 7 단계 참고)

16. 8c단계로 이동하여 교환하여 내보낸 페지식별자를 페지표입구점에 넣고 페지틀을 해제한다.

try_to_swap_out()함수는 입출력자료전송의 활성화를 위해 rw_swap_page()를 직접 호출하지 않는다. 대신 필요하다면 페지를 교환캐쉬에 삽입하고 페지를 불결한것으로 표시한다. 뒤에 나오는 《shrink_cache()함수》에서 보지만 핵심부는 불결한 페지를 소유한 address_space객체의 writepage메쏘드를 호출해서 주기적으로 디스크캐쉬를 흘리기하여 디스크에 쓴다.

앞서《교환캐쉬》에서 설명한것처럼 교환캐쉬에 속한 페지의 address_space 객체는 swapper_space에 저장된 특수한 객체이다. writepage메쏘드는 swap_wfitepage()함수가 실현하며 이 함수는 다음을 수행한다.

1. 페지가 어떤 프로쎄스의 페지표에도 포함되여있지 않음을 검사한다. 아무 표에도 포함되여있지 않으면 페지를 교환캐쉬에서 제거하고 교환페지슬로트를 해제한다.

2. 그렇지 않으면 페지에 대해 WRITE 명령을 지정하여 rw_swap_page()를 호출한다.(《rw_swap_page()함수》 참고)

6. 페지교환하여 넣기

프로쎄스가 자기의 주소공간에 있는 페지중에서 디스크에 교환하여 내보낸 페지를 참조하려고 하면 교환하여 넣기를 해야 한다. 페지오유례외조종기는 다음 조건을 만족할 때 교환하여 넣기연산을 시작한다.(《주소공간안의 잘못된 주소 처리하기》 참고)

・례외를 발생시킨 주소를 포함하는 페지가 유효하다. 즉 이 페지가 현재프로쎄스의 기억기령역에 속한다.

・페지가 기억기에 없다. 즉 페지표입구점의 Present기발이 지워졌다.

・페지에 대응하는 페지표입구점이 NULL이 아니다. 즉 교환하여 내보낸 페지식별자를 가지고있다.

《 요구페지화 》 에서 설명한것처럼 do_page_fault()례외조종기가 호출한 handle_pte_fault()함수는 페지표입구점이 NULL이 아닌지 검사한다. NULL이 아니면 해당 페지를 교환하여 넣기하려고 do_swap_page()함수를 호출한다.

1) do_swap_page()함수

do_swap_page()함수는 다음변수를 받는다.

mm

페지오유례외를 발생시킨 프로쎄스의 프로쎄스서술자주소

vma

address를 포함하는 령역의 기억기령역서술자주소

address

례외를 발생시킨 선형주소

page_table

address를 배치하는 페지표입구점주소

orlg_pte

address를 배치하는 페지표입구점의 내용

write_access

접근요청이 읽기요청인지, 쓰기요청인지 나타내는 기발

다른 함수들과 달리 do_swap_page()함수는 절대로 0을 반환하지 않는다. 페지가 이미 교환캐쉬에 있으면 1을 반환하고 페지를 이미 교환령역에서 읽었으면 2를 반환하고 교환하여넣기를 실행하는 도중에 오유가 발생했으면 -1을 반환한다. 이 함수는 다음 단계를 수행한다.

1. 기억기서술자의 page_table_lock스핀잠그기를 해제한다.(이 함수를 호출한 handle_pte_fault()에서 얻은 잠그기이다.)

2. orig_pte에서 교환하여내보낸 페지식별자를 얻는다.

3. lookup_swap_cache()를 호출하여 교환하여 내보낸 페지식별자에 대응하는 페

지가 이미 교환캐쉬에 있는지 검사한다. 페지가 교환캐쉬에 있으면 6단계로 이동한다.

4. swapin_readahead()함수를 호출하여 교환령역에서 최대 2n페지의 그룹을 읽는다. 여기서 n은 page_cluster변수에 저장되여있고 보통 3이다. read_swap_cache_async()함수를 호출하여 각 페지를 읽는다.

5. read_swap_cache_async()를 다시 호출하여 페지오유를 발생시킨 프로쎄스가 접근한 페지만 교환하여 넣기한다. 이 단계는 불필요하게 중복된것처럼 보이지만 실제로는 그렇지 않다. swapin_readahead()함수는 page_cluster가 0으로 설정되였거나 결함이 있는 페지슬로트(SWAP_MAP_BAD)를 포함하는 페지의 그룹을 읽으려다 요청한 페지를 읽는데 실패할수도 있다. 만약 swapin_readahead()가 성공했으면 다시 read_swap_cacheasync()를 호출하더라도 페지를 교환캐쉬에서 발견하고 빨리 함수를 끝낼수 있다.

6. 여러 시도에도 불구하고 요청한 페지가 교환캐쉬에 추가되지 않으면 이 프로쎄스의 복제를 처리하는 다른 핵심부조종경로가 이미 요청한 페지를 교환하여 넣기했을수도 있다. 이 경우를 검사하기 위해 림시로 page_table_lock 스핀잠그기를 얻어서 page_table이 가리키는 입구점을 orig_pte와 비교한다. 이것들이 다르면 페지를 이미 다른 핵심부스레드가 교환하여 넣기한것이므로 1을 반환한다. 그렇지 않으면 -1을 반환한다.(실패)

7. 여기에 이르면 페지가 교환캐쉬에 있다. mark_page_acccssed()를 호출하고(뒤에 나오는 《LRU목록》 참고) 페지에 잠그기를 건다.

8. page_table_lock스핀잠그기를 얻는다.

9. 이 프로쎄스의 복제를 처리하는 다른 핵심부조종경로가 요청한 페지를 교환하여 넣기했는지 검사한다. 이 경우 page_table_lock스핀잠그기를 해제하고 페지의 잠그기를 풀고 1을 반환한다.

10. swap_free()를 호출하여 입구점(entry)에 대응하는 페지슬로트의 사용계수기를 감소시킨다.

11. 교환캐쉬가 최소한 50% 찼는지 검사한다(nr_swap_pages가 total_swap_pages의 절반보다 작은 경우). 검사한것이 맞으면 오유를 일으킨 프로쎄스(또는 이 프로쎄스의 복제)만이 이 페지를 소유하고있는지 검사한다. 이 경우 페지를 교환캐쉬에서 제거한다.

12. 프로쎄스의 기억기서술자의 rss마당을 증가시킨다.

13. 페지의 잠그기를 해제한다.

14. 페지표입구점을 갱신하여 프로쎄스가 페지를 발견할수 있도록 한다. 함수는 요청한 페지의 물리적주소와 기억기령역의 vm_page_prot마당의 보호비트를 page_table이 가리키는 페지표입구점에 기록한다. 그리고 오유를 일으킨 접근이 쓰기이고 오유를 일으킨 프로쎄스가 페지의 유일한 소유자면 함수는 불결한 기발과 읽기/쓰기기발도 설

정하여 불필요한 《쓰기복사》오유를 방지한다.

15. mm->page_table_lock스핀잠그기를 해제하고 1 또는 2를 반환한다.

7. 페지틀 회수하기

Linux의 가상기억기보조체계는 전체 핵심부에서 가장 복잡하고 성능에 중요한 구성요소이다.

앞에서 핵심부가 어떻게 사용중 또는 여유페지틀을 관리하여 동적기억기를 처리하는지 설명하였다. 또한 사용자방식의 각 프로쎄스는 자신의 선형주소공간이 있으므로 페지틀이 프로쎄스에 가능한 늦게 할당됨을 설명하였다. 그리고 느린 블로크장치의 자료를 캐쉬하기 위해 동적기억기를 사용하는것도 설명하였다.

여기서는 페지틀회수를 설명하면서 기상기억기보조체계에 관한 설명을 마친다. 앞에서 살펴본것처럼 캐쉬체계는 페지틀을 계속 획득하지만 해제하지는 않는다. 사실 캐쉬체계는 프로쎄스가 언제 어떤 캐쉬자료를 사용할지 알수 없으므로 어떤 캐쉬페지를 해제해야 할지 알수가 없다. 그리고 앞에서 설명한 《요구페지화》기법으로 하여 사용자방식프로쎄스는 실행과정에서 페지틀을 획득하게 되지만 요구페지화에도 프로쎄스가 더는 사용하지 않는 페지틀을 해제하도록 할 방법이 없다. 이 문제의 해결책이 페지틀 회수기법이다.

핵심부개발자에게 가장 어려운것은 사용할 페지틀이 더는 남아있지 않은 상태이다. 이 상태가 발생하면 핵심부는 쉽게 해결할수 없는 련속된 기억기요청에 빠지게 된다. 페지틀을 해제하기 위해서 핵심부는 반드시 그 자료를 디스크에 기록해야 한다. 그러나 핵심부가 이 연산을 처리하려면 다른 페지틀이 필요하다.(례를 들어 입출력자료전송을 위한 완충기머리부를 할당해야 한다.) 여유페지틀이 없기때문에 어떤 페지틀도 해제할수 없다. 이 상태에서 해결책은 한가지뿐이다. 사용자방식프로쎄스 하나를 희생해서 해당 프로쎄스가 사용하던 페지틀들을 회수하는것이다. 물론 이 해결책이 체계파괴를 막을수는 있겠지만 사용자립장에서는 아주 불쾌할것이다. 페지틀 회수의 목표는 여유페지틀을 몇개 유지하여 핵심부가 《기억기 부족》상태에서 안전하게 회복하는것이다. 이를 위해서 디스크캐쉬를 아무렇게나 지우거나 사용자방식프로쎄스에 너무 많은 부담을 주면 안된다. 그렇게 하면 체계성능이 너무도 나빠질것이다. 사실 가상기억기보조체계개발의 가장 어려운 부분이 탁상형(이 경우에 기억기요청은 제한적이다.)과 자료기지봉사기같은 고성능봉사기(기억기요청이 아주 많다.)량쪽에서 만족할만한 성능을 나타내는 알고리듬을 찾는것이다.

다행히도 좋은 페지틀회수알고리듬을 찾는것은 리론의 도움이 거의없이 실험에 의존하는일이다. 이 상태는 프로쎄스의 동적인 우선순위를 결정하는 요소를 평가하는것과 비슷하다. 기본목적은 파라메터를 조종해서 좋은 성능을 얻는것이고 왜 잘 동작하는지 알아내는것이 아니다. 《이 새로운 접근방법을 시도해서 어떤 일이 벌어지는지 보자》는 식이 될 때가 많다. 실험적인 접근방법의 부작용은 코드변화가 쉽게 일어난다는것이

다.(심지어 안정판본인 짝수판본에서도 일어난다.)

1) **페지틀회수 알고리듬의 개요**

자세히 설명하기 전에 Linux의 페지틀 회수를 간단히 보자.(나무의 잎에 너무 가까이 다가가면 전체 숲을 놓칠수 있다.)

페지틀은 두가지 방법으로 회수할수 있다.

・캐쉬(기억기캐쉬 또는 디스크캐쉬)에 있는 사용되지 않는 페지틀을 회수한다.

・프로쎄스의 기억기령역 또는 IPC공유기억기령역(《IPC공유기억기》참고)에 속하는 페지를 회수한다.

물론 알고리듬은 페지틀의 차이를 고려해야 한다. 례를 들면 디스크캐쉬보다 기억기캐쉬에서 페지틀을 회수하는것이 더 바람직하다. 디스크캐쉬의 페지는 블로크디스크장치에서 상대적으로 많은 비용을 들여 읽어들인 자료를 포함하기때문이다.

그리고 알고리듬은 각 페지틀에 대한 접근회수를 관리해야 한다. 어떤 페지에 오래동안 접근하지 않았다면 가까운 미래에 접근할 확률이 낮다. 반면에 최근 어떤 페지에 접근하였다면 계속 접근할 확률이 높다. 이것도 《하드웨어캐쉬》에서 설명한 지역성(locality)규칙의 응용레다.

따라서 페지틀 회수알고리듬은 여러가지 경험규칙을 섞은것이다.

・캐쉬검사순서에 대한 주의깊은 선택

・오래된 순서에 따른 페지의 순서(최근에 사용한 페지보다 최근에 사용하지 않은 페지를 먼저 해제한다.)

・페지상태에 따른 페지의 구분(례를 들면 불결한 페지보다 불결한것이 아닌 페지를 교환하여 내보내는것이 더 좋다. 불결한것이 아닌 경우에는 디스크에 쓸 필요가 없다.)

페지틀회수를 시작하는 주요함수는 try_to_free_pages()이다. 핵심부가 기억기할당에 실패할 때마다 이 함수를 호출한다. 례를 들면 다음과 같다.

・grow_buffers()함수가 새로운 완충기페지할당에 실패하거나 create_buffers()함수가 완충기페지를 위한 완충기머리부할당에 실패한 경우.(《완충기페지》와 《getblk()함수》 참고)

이 경우 핵심부는 free_more_memory()를 호출하고 이 함수가 try_to_free_pages()를 호출한다.

・page_alloc()함수가 주어진 기억기지역(zone)목록에서 페지틀그룹할당에 실패한 경우.(《형제체계알고리듬》참고) 모든 기억기구역서술자는 pages_min 마당을 포함한다. 이 마당은 《기억기부족》상태에 대처하기 위해 여유상태로 남아있어야 하는 페지틀의 수를 나타낸다. 목록의 어떤 구역에도 여유페지틀의 최소량을 만족하면서 요청을 만족하기에 충분한 여유기억기가 없다면 핵심부는 balance_classzone()함수를 호출하고 이 함수가 try_to_free_pages()함수를 호출한다.

・kswapd핵심부스레드가 어떤 기억기지역의 여유페지틀수가 page_low 이하로 떨

어진 사실을 발견했을 때.(뒤에 나오는《 kswapd 핵심부스레드》 참고)

try_to_free_pages()함수의 핵심은 shrink_caches()함수이다. 이 함수는 변수로 goal을 받는다. goal은 회수할 페지들의 수이다. 이 함수는 이 목표에 도달하면 즉시 중단한다. shlink_caches()의 일을 돕기 위해서 동적기억기의 모든 페지는 활성목록과 비활성목록이라는 두 그룹으로 나눈다. 이것들을 LRU목록이라고 부르기도 한다. 활성목록은 최근에 사용한 페지를 포함하고 비활성목록은 한동안 사용하지 않은 페지를 포함한다. 마땅히 비활성목록에서 페지를 회수한다. 종종 두 목록사이에서 페지가 이동하기도 한다. shrink_caches()함수는 다음 함수들을 차례로 호출한다.

kmem_cache_reap()

스랩캐쉬에서 빈(empty)스랩를 제거한다.

refill_inactive()

페지를 활성목록에서 비활성목록으로 또는 반대로 옮긴다.

shrink_cache()

페지캐쉬에 포함된 비활성페지를 디스크에 기록하여 페지들을 해제하려고 한다.

shrink_dcache_memory()

등록부입구점캐쉬에서 입구점을 제거한다.

shrink_cache_memory()

i마디캐쉬에서 입구점을 제거한다.

이제 페지틀회수알고리듬의 여러 구성요소를 자세히 알아보자.

2) LRU목록

페지의 활성목록과 비활성목록은 페지틀회수알고리듬의 핵심자료구조이다. 이 두 2중련결목록의 머리부는 각각 active_list와 inactive_list변수에 저장된다. nr_active_pages와 nr_active_pages 변수는 두 목록의 페지수를 저장한다. pagemap_1ru_lock 스핀잠그기는 SMP체계에서 두 목록에 대한 동시접근을 방지한다.

어떤 페지가 LRU목록에 속하면 페지서술자의 PG_lru기발을 설정한다. 그리고 페지가 활성목록에 속하면 PG_active기발을 설정하고 비활성목록에 속하면 PG_active기발을 지운다. 페지서술자의 lru 마당은 LRU목록의 다음과 이전요소를 가리키는 지적자를 저장한다.

LRU목록을 처리하기 위한 다음과 같은 보조함수와 마크로가 있다.

add_page_to_active_list

PG_active기발을 설정하고 페지를 활성목록의 머리부에 추가하고 nr_active_pages를 증가시킨다.

add_page_to_inactive_list

페지를 비활성목록의 머리부에 추가하고 nr_inactive_pages를 증가시킨다.

del_page_from_active_list

페지를 활성목록에서 제거하고 PG_active기발을 지우고 nr_active_pages를 감소시킨다.

del_page_from_inactive_list

페지를 비활성목록에서 제거하고 nr_inactive_pages를 감소시킨다.

activate_page_nolock()과 activate_page()

페지가 비활성목록에 있으면 del_page_from_inactive_list와 add_page_to_activc_list를 호출하여 활성목록으로 옮긴다. active_page()함수는 페지를 옮기기 전에 pagemap_lru_1ock스핀잠그기를 얻는다.

1ru_cache_add()

페지가 LRU목록에 없으면 PG_1ru기발을 설정하고 pagemap_lru_1ock 스핀잠그기를 얻고 add_page_to_inactive_list를 호출하여 페지를 비활성목록에 추가한다.

__lru_cache_del()과 1ru_cache_del()

페지가 LRU목록에 있으면 PG_1ru기발을 지우고 PG_active기발의 값에 따라 del_page_from_activc_list 또는 del_page_from_inactive_list를 실행한다. 1ru_cache_del()함수는 페지를 제거하기 전에 pagemap_lru_1ock 스핀잠그기를 얻는다.

◦ LRU목록 사이의 페지이동

핵심부는 최근에 접근한 페지들을 활성목록으로 관리하고 회수할 페지틀을 찾아볼 때 활성목록을 탐색하지 않는다. 반대로 핵심부는 오래동안 접근하지 않은 페지들을 비활성목록으로 관리한다. 물론 페지를 접근함에 따라 페지를 비활성목록에서 활성목록으로 또 반대로 옮겨줘야 한다.

물론 페지의 활성과 비활성이라는 두가지 상태가 모든 접근패턴을 나타내는데 충분하지는 않다. 례를 들어 기록프로쎄스가 자료를 페지에 한시간에 한번씩 기록한다고 하자. 이 페지는 대부분의 시간에는 비활성상태지만 한번 접근하면 활성상태로 바뀌고 앞으로 한 시간동안 접근하지 않을것임에도 불구하고 대응하는 페지틀을 회수하지 못하게 한다. 물론 핵심부가 사용자방식프로쎄스의 행동을 예측할수 있는 방법이 없으므로 이 문제에 대한 일반화된 해결책은 없다. 그러나 한번 접근할 때마다 페지의 상태가 바뀌는 것은 문제가 있어보인다.

어떤 페지를 비활성목록에서 활성목록으로 옮기는데 필요한 접근회수를 두배로 하기 위해 페지서술자의 PG_referenced 기발을 사용한다. 이 기발은 어떤 페지를 활성목록에서 비활성목록으로 옮기는데 필요한 잘못된 접근의 수를 두배로 하는데도 사용된다. 례를 들어 비활성목록의 어떤 페지의 PG_referenced 기발이 0으로 설정되여있다고 하자. 처음으로 페지에 접근하면 이 기발의 값을 1로 바꾸지만 페지를 계속 비활성목록에 놓아둔다. 두번째로 페지에 접근하면 이 기발이 설정되여있음을 알게 되고 페지를 활성목록으로 옮긴다. 그러나 두번째 접근이 첫번째 접근이후 일정한 시간간격안에 일어나지 않으면 페지틀회수알고리듬은 PG_reference 기발을 지운다.

그림 4-24에서 볼수 있는것처럼 핵심부는 mark_page_accessed()와 refill_inactive()함수를 사용하여 페지를 LRU목록사이에서 옮긴다. 그림에서 PG_active기발은 페지가 어떤 LRU목록에 포함되였는지 나타낸다.

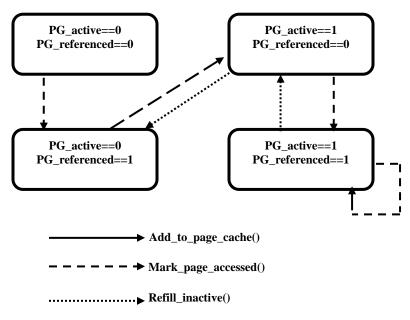



그림 4-24. LRU 목록사이의 페지이동

핵심부는 어떤 페지에 접근한것을 표시하기 위해 언제나 mark_page_accessed()함수를 사용한다. 핵심부가 사용자방식프로세스파일체계 계층 또는 장치구동프로그람이 어떤 페지를 참조한다고 판단하면 언제나 이 함수를 호출한다. 례를 들어 다음과 같은 경우 mafk_page_accessed()를 호출한다.

· 요구페지화(demand pageing)에 의해 닉명페지를 적재할 때.(《요구페지화》에서 설명한 do_anonymous_page()함수에 의해 실행된다.)

· 디스크에서 블로크를 읽을 때.(《블로크와 페지입출력연산》에서 설명한 bread()함수에 의해 실행된다.)

· 요구페지화에 의해 기억기배치파일의 페지를 적재할 때.(《기억기배치의 요구페지화》에서 설명한 filemap_nopage()함수에 의해 실행된다.)

· 파일에서 한 페지의 자료를 읽을 때.(《파일에서 읽기》에서 설명한 do_genefic_file_read()함수에 의해 실행된다.)

· 페지를 교환하여 넣기할 때.(앞서 살펴본 《do_swap_page()함수》참고)

· 핵심부가 탐색중에 페지표입구점의 Accessed기발이 설정된것을 발견했을 때.(앞서 살펴본 《try_to_swap_out()함수》참고)

· 핵심부가 디스크장치에서 한 페지의 자료를 읽을 때.(ext2_get_page()함수에 의

해 실행된다.)

mark_page_accessed()함수는 다음과 같은 부분코드를 실행한다.

```
if(pageActive(page) || !PageReferenced(page))
SetPageReferenced(page);
else{
activate_page(page);
ClearPageReferenced(page);
  }
```

그림 4-24에서 볼수 있는것처럼 이 함수는 PG_referenced기발이 이미 설정되여있을 때에만 비활성목록에서 활성목록으로 페지를 옮긴다.

핵심부는 주기적으로 refill_inactive()함수를 실행하여 활성목록에 있는 페지의 상태를 검사한다. 이 함수는 활성목록의 끝(목록에서 가장 오래된 페지)에서 시작하여 각 페지의 PG_referenced기발이 설정되여있는지 검사한다. 이 함수는 기발이 설정된 페지가 있으면 해당 페지의 기발을 지우고 페지를 활성목록의 맨 처음으로 옮긴다. 기발이 설정되여있지 않으면 페지를 비활성목록의 맨 처음으로 옮긴다. 이 부분에 해당하는 코드는 다음과 같다.

```
if(PageReferenced(page)){
ClearPageReferenced(page);
list_del(&page->lru);
list_add(&page->lru, &active_list);
}else{
del_page_from_active_list (page);
add_page_to_inactive_list (page);
SetPageReferenced(page);
}
```

refill_inactive()함수는 비활성목록의 페지를 탐색하지 않는다. 따라서 페지가 비활성목록에 있는동안에는 페지의 PG_referenced기발은 설정된 상태로 유지된다.

○ try_to_free_pages()함수

try_to_free_pages()는 페지들의 회수를 시작하는 주요함수이다. 이 함수는 다음과 같은 변수를 받는다.

classzone

회수할 페지들을 포함하고있는 기억기지역

gfp_mask

기발의 집합으로서 그 의미는 alloc_pages()함수의 경우와 동일하다.(《페지들의 요청과 해체》참고)

order
사용되지 않는다.
이 함수의 목적은 페지를 SWAP_CLUSTER_MAX(보통 32)개를 해제하는것이며 이것을 위해 shrink_caches()함수를 우선순위를 높여가며 반복해서 호출한다. try_to_free_pages()함수는 기본적으로 다음 부분코드와 같다.

```
for (priority = DEF_PRIORITY; priority >= 0; priority--) {
        sc.nr_mapped = read_page_state(nr_mapped);
        sc.nr_scanned = 0;
        sc.nr_reclaimed = 0;
        sc.nr_congested = 0;
        sc.priority = priority;
        shrink_caches(zones, &sc);
        shrink_slab(sc.nr_scanned, gfp_mask, lru_pages);
        if (reclaim_state) {
            sc.nr_reclaimed += reclaim_state->reclaimed_slab;
            reclaim_state->reclaimed_slab = 0;
        }
        if (sc.nr_reclaimed >= SWAP_CLUSTER_MAX) {
            ret = 1;
            goto out;
        }
        total_scanned += sc.nr_scanned;
        total_reclaimed += sc.nr_reclaimed;

        if (total_scanned > SWAP_CLUSTER_MAX + SWAP_CLUSTER_
MAX/2) {
            wakeup_bdflush(laptop_mode ? 0 : total_scanned);
            sc.may_writepage = 1;
        }

        if (sc.nr_scanned && priority < DEF_PRIORITY - 2)
            blk_congestion_wait(WRITE, HZ/10);
    }
    if ((gfp_mask & __GFP_FS) && !(gfp_mask & __GFP_NORETRY) &&
            sc.nr_congested < SWAP_CLUSTER_MAX)
```

out_of_memory(gfp_mask);

현재프로쎄스의 PF_NOIO 기발이 설정되여있으면 try_to_free_pages()는 gfp_mask 변수의 __GFp_IO, __GFP_HIGHIO, __GFP_FS 비트를 지운다. 핵심부는 페지틀회수알고리듬이 입출력자료전송을 시작하지 않게 하려고 PF_NOIO기발을 설정한다. 현재는 사용자방식프로쎄스가 정규파일을 디스크블로크구획처럼 사용할수 있도록 하는 순환장치구동프로그람의 핵심부스레드에서만 사용한다.

순환은 최대 DEF_PRIORITY 회수만큼(보통 6회) 반복하며 매번 감소하는 순환색인값을 shrink_caches()함수에 전달한다. 더 작은 값이 더 높은 우선순위를 나타내므로 매번 shrink_caches()는 페지틀을 더 심하게 회수하려고 한다.

DEF_PRIORRITY 회수만큼의 반복이 페지틀 SWAP_CLUSTER_MAX개를 회수하는데 충분하지 않으면 핵심부는 심각한 상태이다. 마지막 해결방법은 사용자방식프로쎄스 하나를 제거해서 해당 프로쎄스의 모든 페지틀을 회수하는것이다. out_of_memory()함수가 이 연산을 수행한다. 간단히 말해서 초사용자권한이 없고 직접 입출력연산을 수행하지 않는 프로쎄스중에서 실행시간이 가장 짧은 프로쎄스를 선택하여 제거한다.(《직접입출력전송》 참고)

◦ **shrink_caches( )함수**

shrink_caches()함수는 여러 보조함수를 고정된 순서로 호출하여 여러 기억기보조체계에서 페지틀을 회수한다. 호출되는 함수중에는 shrink_cache()가 있으며 호출하는 함수 shrink_caches()와 혼동하지 말아야 한다. shrink_caches()함수는 다음과 같은 변수를 받는다.

classzone

회수할 페지틀을 포함하는 기억기지역.

priority

이번 호출의 우선순위. 페지틀 회수를 얼마나 세게 할것인지 나타낸다.

gfp_mask

기억기할당기발. 해제할 페지틀의 류형과 수행과정에서 핵심부가 시용할수 있는 기법을 나타낸다.(현재프로쎄스를 차단하거나 입출력전송을 시작하도록 하는 등)

nr_pages

목표로 하는 해제할 페지틀의 수

이 함수는 nr_pages와 실제로 회수한 페지틀수의 차이를 반환한다. nr_pages보다 더 많은 페지틀을 해제했으면 0을 반환한다. 이 함수는 다음과 같은 작업을 수행한다.

1. kmem_cache_reap()을 호출하여 스랩할당자캐쉬로부터 페지틀을 회수한다.(《캐쉬에서 스랩해제하기》 참고)

2. kmem_cache_reap()이 최소 nr_pages이상의 페지틀을 해제하는데 성공했으면 0을 반환한다.

3. refill_inactlve()함수를 호출하여 일부 페지를 활성목록에서 비활성목록으로 옮긴다. 앞서 《LRU목록》에서 설명한것처럼 refill_inactive()는 활성목록의 끝에 있는 페지들의 PG_referenced 기발을 지우고 이전 shrink_caches()실행이후 접근한 사실이 없는 페지들을 옮긴다. 옮길 페지의 수를 refil_inactive()에 변수로 전달한다. 다음과 같은 수식을 사용한다.

ratio = nr_pages * nr_active_pages / ((nr-inactive_pages + 1) * 2);

이 수식의 근거는 활성목록의 크기를 대략 페지캐쉬크기의 2/3정도로 유지하는것이다.(역시 실험적인 수식이다.)

4. shrink_cache()를 호출하여 비활성목록에서 페지틀 nr_pages개를 회수하려고 시도한다. 함수가 요청한 수의 페지틀을 회수하는데 성공하면 0(요청한 수의 페지틀을 회수했음)을 반환한다.

5. 여기에 이르면 shrink_caches()는 현재실행에서 목표에 도달할 가능성이 없다. 그렇지만 여러 디스크캐쉬에서 작은 객체들을 해제하려고 시도하여 이후의 함수호출에서 작은 객체들을 저장하고있는 페지틀을 해제하는데 성공할수 있게 한다. 따라서 함수는 shrink_dcache_memory()를 호출하여 등록부입구점객체를 등록부입구점캐쉬에서 제거하도록 한다.(뒤에 나오는 《등록부입구점과 i마디 캐쉬에서 페지틀 회수하기》참고)

6. shrink_icache_memory()를 호출하여 i마디캐쉬에서 i마디객체를 제거한다.(뒤에 나오는 《등록부입구점과 i마디캐쉬에서 페지를 회수하기》 참고)

7. 핵심부가 디스크할당(quota)를 지원하는 경우 shrink_dqcache_memory()를 호출하여 디스크할당캐쉬에서 객체를 제거한다.(디스크할당에 대해서는 생략한다.)

8. 지금까지 해제한 페지틀의 수를 반환한다.

○ **shrink_cache()함수**

shrink_cache()함수는 shrink_caches()와 같이 nr_pages, classzone, gfp_mask, priority를 변수로 받는다. 이 함수는 비활성목록에서 회수할 페지틀을 찾는다. 최근에 삽입한 요소가 목록의 머리부쪽에 있으므로 함수의 탐색은 목록끝부터 역방향으로 이루어진다,

함수는 목적을 이루기 위해 다음과 같이 동작한다.

・아무 프로쎄스에도 속하지 않는 페지틀을 형제체계로 물려준다.

・비활성목록의 탐색한 부분에서 프로쎄스에 속한 페지들이 대부분이면 프로쎄스에 속한 페지를 교환하여 내보낸다.

priority변수는 이번의 shrink_cache()호출에서 탐색할 비활성목록의 크기를 조종한다. priority가 6(DEF_PRIORITY, 가장 낮은 우선순위)이면 함수는 목록의 1/6을 탐색한다. 우선순위가 작아지면서 함수는 목록의 더 많은 부분을 탐색하며 1(가장 높은 우선순위)이 되면 전체 목록을 탐색한다.

함수는 시작부분에서 pagemap_lru_lock 스핀잠그기를 얻고 비활성목록의 페지를

역방향으로 순환을 돌면서 우선순위가 나타내는 수의 요소를 탐색할 때까지 계속한다. 탐색하는 페지에 대해 다음과 같은 작업을 수행한다.

1. 현재프로쎄스의 need_resched 마당이 설정되여있으면 pagemap_lru_lock스핀잠그기를 해제하여 림시로 CPU를 풀어주고 schedule()을 호출한다. 다시 실행하게 되면 다시 스핀잠그기를 얻고 계속한다.

2. 페지를 비활성목록의 끝에서 머리부부분으로 옮긴다. 이렇게 함으로써 이후의 shrink_cache()호출에서 비활성페지를 돌아가면서(round_robin) 검사하도록 한다.

3. 페지의 사용계수기가 0인지 검사한다. 0이면 목록의 끝에서 다음페지에 대해 계속한다. 사용계수기가 0인 페지는 형제체계에 속해있는것이 바람직하지만 페지들을 해제하기 위해서는 먼저 사용계수기를 감소시키고 페지들을 형제체계로 반납한다. 따라서 페지들회수알고리듬에서 해제된 페지를 볼수 있는 짧은 시간간격이 존재한다.

4. 페지가 classzone변수가 나타내는 기억기지역에 속하는지 검사한다. 속하지 않으면 이 페지에 대해 동작을 멈추고 목록끝에 있는 다음페지에 대해 계속한다.

5. 페지가 완충기페지가 아니고 사용계수기가 1보다 크거나 페지가 디스크에 영상을 가지고있지 않는지(mapping 마당이 NULL) 검사한다. 이 경우 다른 프로쎄스가 페지를 참조하고있다는 사실을 사용계수기에서 나타내므로 페지들을 형제체계로 반납할수 없다. 함수는 다음 단계를 수행한다.

a. 탐색한 페지중에 해제할수 없는 페지의 수를 나타내는 지역계수기를 증가시킨다.

b. 계수기가 경계값을 넘으면 pagemap_lru_lock 스핀잠그기를 해제하고 swap_out()을 호출하여 일부프로쎄스페지를 교환하여 내보내도록 한다. (《페지를 교환하여 내보내기》참고) 그리고 해제해야 하는 페지들의 수를 반환한다. 경계값은 다음 두 값중 작은 값이다. 하나는 탐색할 페지수의 1/10이고 다른 하나는 210-prority * 해제할 페지들수(nr_pages 변수)다.

c. 그렇지 않고 계수기가 경계값을 넘지 않으면 함수는 비활성목록의 끝에서 다음 페지에 대해 계속한다.

6. 함수가 여기에 이르면 페지들을 형제체계로 반환할수 있다. 함수는 페지에 잠그기를 걸려고 시도한다. 페지의 PG_locked 기발이 이미 설정되여있으면 다음단계를 수행한다.

a. 페지의 PG_launder기발과 gfp_mask 변수의 __GFP_FS비트가 둘 다 설정되여 있으면 wait_on_page()를 호출하여 페지의 잠그기가 해제될 때까지 잠든다. PG_launder기발은 shrink_cache()함수자신이 시작하게 한 입출력자료전송에 페지가 포함되면 언제나 설정된다.

b. 비활성목록의 끝에서 다음페지에 대해 계속한다.

7. 이제 페지에 잠그기가 걸려있다. 페지가 불결한지(PG_dirty기발이 설정되여있는지), 페지가 디스크에 영상을 가지고있는지(mapping마당이 NULL이 아닌지), 페지

캐쉬만이 페지를 소유하고있는지 즉 페지틀을 실제 해제할수 있는지 검사한다. 모든 조건을 만족하고 gfp_mask 변수의 __GFP_FS 비트가 설정되여있으면 함수는 다음을 수행하여 디스크영상을 갱신한다.

a. PG_dirty 기발을 지운다 .

b. PG_launder기발을 설정하여 이후의 shrink_cache()호출은 입출력자료전송의 완료를 기다리도록 한다.

c. 페지사용계수기를 증가시키고(안전장치) pagemap_lru_lock 스핀잠그기를 해제한다.

d. 페지의 address_space 객체의 writepage 메쏘드를 호출한다. 《불결한 기억기배치페지를 디스크에 흘리기》에서 설명한것처럼 이 메쏘드는 페지내용을 디스크로 입출력자료전송하는것을 활성화한다.

e. 페지사용계수기를 감소시키고 pagemap_lru_lock 스핀잠그기를 다시 획득한다.

f. 비활성목록의 끝에서 다음페지에 대해 계속한다.

8. 페지가 완충기페지면(buffers 마당이 NULL이 아님) 함수는 페지에 포함된 완충기를 해제하려고 시도한다.

다음과 같은 단계를 실행한다 .

a. pagemap_lru_lock스핀잠그기를 해제하고 페지사용계수기를 증가시킨다.(안전장치)

b. try_to_release_page()함수를 호출한다. 이 함수는 다음을 수행한다.

- 대응하는 address_space 객체의 releasepage메쏘드가 정의되여있으면 이 메쏘드를 실행하여 기록형파일체계에서 완충기에 련결된 조종자료를 해제하도록 한다.

- 완충기캐쉬만이 페지의 완충기를 참고하고있으면 tty_to_free_buffers()함수를 호출하여 해제하도록 한다.(《완충기페지》참고)

c. try_to_release_page()가 페지의 모든 완충기를 해제하는데 실패하면 함수는 페지의 잠그기를 해제하고 8a에서 증가시킨 사용계수기를 감소시키고 비활성목록의 끝에서 다음페지에 대해 계속한다.

d. 그렇지 않고 try_to_release_page()가 페지의 모든 완충기를 해제하는데 성공하면 함수는 페지틀자체를 해제하려고 시도한다. 특히 페지가 닉명이면(즉 디스크에 영상이 없으면)함수는 pagemap_lru_lock 스핀잠그기를 획득하고 페지의 잠그기를 풀고 페지를 비활성목록에서 제거하고 페지틀을 형제체계에 반환한다. 함수가 목표로 하는 수의 페지틀을 해제했으면 스핀잠그기를 해제하고 0을 반환한다. 그렇지 않으면 9단계에서 계속한다.

e. 완충기페지가 디스크영상을 가지고있으므로 페지캐쉬에 포함되여있다. 8a 단계에서 증가시킨 사용계수기를 감소시키고 pagemap_lru_lock 스핀잠그기를 얻는다. 그리고 다음단계에서 계속한다.

9. pagecache_lock 스핀잠그기를 획득한다.

10. 페지가 디스크에 영상이 없거나 페지를 참조하는 프로쎄스가 있으면 pagecache_lock스핀잠그기를 해제하고 페지의 잠그기를 해제하고 5a 단계로 이동한다.

11. 함수가 여기에 이르면 페지는 디스크에 영상이 있고 페지를 참조하는 프로쎄스가 없으며 완충기를 가지고있지 않으므로 해제할수 있다. 페지가 불결한지(PG_dirty기발이 설정되였는지) 검사한다. 불결하면 페지들을 해제할수 없다. 해제하면 자료를 잃게 된다. 함수는 pagecache_lock 스핀잠그기를 해제하고 비활성목록의 끝에서 다음페지에 대해 계속한다.

12. 함수가 여기에 이르면 페지는 디스크에 영상이 있으며 해제할수 있고 불결한것이 아니다. 따라서 실제로 페지들을 해제할수 있다. 페지가 교환캐쉬에 속해있으면 함수는 index 마당에서 교환하여내기페지식별자를 얻고 delete_from_swap_cache()를 호출하여 교환캐쉬에서 페지서술자를 제거하고 pagecache_lock 스핀 잠그기를 해제하고 swap_free()를 호출하여 페지슬로트의 사용계수기를 감소시킨다.

13. 그렇지 않으면 페지가 교환캐쉬에 속하는지 검사한다. 속하지 않으면 remove_inode_page()를 호출하여 페지캐쉬에서 제거하고(《페지캐쉬처리함수》 참고) pagecache_lock 스핀잠그기를 해제한다.

14. __lru_cache_del()을 호출하여 비활성목록에서 페지를 제거한다.

15. 페지의 잠그기를 제거한다.

16. 페지들을 형제체계로 반환한다.

17. 함수가 목표로 하는 수의 페지들을 해제했으면 스핀잠그기를 해제하고 0을 되돌이한다. 그렇지 않으면 비활성목록의 끝에서 다음페지에 대해 계속한다.

3) 등록부입구점과 i마디캐쉬에서 페지들회수하기

등록부입구점객체와 i마디객체자체는 크지 않지만 이 객체를 해제하면 련속적인 효과로 여러 자료구조를 해제하여 많은 기억기를 해제할수 있다. 이런 리유로 하여 shrink_caches()함수는 등록부입구점과 i마디캐쉬에서 페지들을 회수하는 전용함수 두개를 호출한다.

○ 등록부입구점캐쉬에서 페지회수하기

등록부입구점캐쉬에서 등록부입구점객체를 제거하기 위해 shrink_dcache_memory() 함수를 호출한다. 물론 어떤 프로쎄스도 참조하지 않는(《등록부입구점객체》에서 사용되지 않는 등록부입구점으로 정의한) 등록부입구점객체만 삭제할수 있다.

등록부입구점캐쉬객체가 스랩할당자를 통해 할당되였으므로 shrink_dcache_memory()함수는 일부 스랩을 비울수 있고 결과적으로 kmem_cache_reap()함수가 일부페지들을 회수할수 있다. 그리고 등록부입구점캐쉬는 i마디캐쉬의 조종기로 동작하므로 등록부입구점객체를 해제하면 대응하는 i마디를 저장한 완충기도 사용되지 않게 되여

shrink_mmap()함수가 대응하는 완충기페지를 해제할수 있게 된다.

shrink_dcache_memory( )함수는 priority와 gfp_mask라는 두 변수를 받아 다음 단계를 수행한다.

1. 핵심부가 파일체계의 디스크자료구조에 대해 연산을 시작하는것이 허용되지 않으면 0을 반환한다.(gfp_mask 변수의 __GFP_IO 비트가 설정되여있지 않음)

2. 그렇지 않으면 사용되지 않는 등록부입구점의 수를 priority 값으로 나눈 값을 변수로 prune_dcache()를 호출한다.

3. kmem_cache_shrink()를 등록부입구점캐쉬에 대해 호출하여 앞단계에서 해제한 객체를 포함하는 프레임을 해제한다.

4. 0을 반환한다.

prune_dcache()함수는 해제할 객체의 수를 나타내는 변수를 받는다. 함수는 요청한 수의 객체를 해제하거나 목록 전체를 탐색할 때까지 목록을 탐색한다. 최근에 참조하지 않은 객체에 대해 prune_one_dentry()를 호출한다.

prune_one_dentry()함수는 다음 연산을 실행한다.

1. 등록부입구점객체를 등록부입구점하쉬표와 부모등록부의 등록부입구점객체목록, 소유자 i마디의 등록부입구점객체목록에서 제거한다.

2. d_input등록부입구점메쏘드가 정의되여있으면 이 메쏘드를, 그렇지 않으면 iput()함수를 호출하여 등록부입구점의 i마디의 사용계수기를 감소시킨다.

3. d_release등록부입구점메쏘드가 정의되여있다면 이 메쏘드를 호출한다.

4. kmem_cache_free()를 호출하여 스랩할당자에서 객체를 해제한다.(《캐쉬에서 객체해제하기》참고)

5. 부모등록부의 사용계수기를 감소시킨다.

◦ **i마디캐쉬에서 페지회수하기**

shrink_icache_memory()함수를 호출하여 i마디캐쉬에서 i마디객체를 제거한다. 조금전에 설명한 shrink_dcache_memory()와 아주 비슷하다. gfp_mask 변수의 __GFP_FS비트를 검사하고 해제할 i마디의 수(즉 사용되지 않는 i마디의 수를 우선순위로 나눈 값)를 변수로 prune_icache()를 호출한다. 끝으로 prune_cache()를 호출하여 페지들을 해제하고 형제체계로 반환하기 위해 kmem_cache_shrink()를 호출한다.

prune_icache()함수는 inode_unused 목록을 탐색하여(《i마디객체》참고) 해제할 i마디를 찾는다. 좋은 후보는 불결한것이 아니고 사용계수기가 0이고 I_FREEING, I_CLEAR, I_LOCK 기발이 설정되여있지 않으며 불결한 완충기목록에 완충기머리부를 가지지 않는것이다. 이와 같은 i마디를 clear_inode()와kmem_cache_free()함수를 호출하여 해제한다.

prune_icache()가 i마디객체를 요청한 수만큼 해제하는데 실패하면 try_to_sync_unused_inodes()함수를 실행하도록 한다. 이 함수는 사용되지 않는 i마

디 일부를 디스크로 흘린다. 이 함수는 차단될수 있으므로 keventd 핵심부스레드에 의해 실행된다.(《핵심부스레드》 참고)

4) kswapd 핵심부스레드

kswapd핵심부스레드는 기억기의 회수를 수행하는 또 다른 핵심부기구이다. 이것이 왜 필요한가? 빈 기억기가 정말로 부족하고 기억기할당요청이 들어왔을 때 try_to_free_pages()를 호출하면 충분하지 않을가?

불행하게도 이런 경우만 있는것이 아니다. 어떤 기억기할당요청은 새치기나 례외조종기를 실행하며 이것들은 여유페지들을 기다리는 현재프로쎄스를 차단하지 못한다. 그리고 어떤 기억기할당요청은 이미 중요한 지원에 대해 배타적인 접근을 획득한 핵심부조종경로에 의해 이루어지므로 입출력자료전송을 활성화할수 없다. 드문 경우지만 모든 기억기할당요청이 이와 같은 핵심부조종경로에 따라 이루어진 경우 핵심부는 영원히 기억기를 비우지 못한다.

또 kswapd는 콤퓨터가 아무 일도 하지 않을 시간에 기억기회수를 수행하여 체계성능을 높인다. 프로쎄스는 페지를 훨씬 빨리 얻을수 있다.

kswapd 핵심부스레드는 어떤 지역(zone)이 어떤 경계수위보다 적은 수의 여유페지들을 포함하고있으면 활성화된다. 핵심부는 더 심한 기억기부족상태를 피하기 위해 페지들 몇개를 회수한다.

《경계수위(warning thresbold)》는 기억기구역서술자의 pages_1ow 마당에 저장되여있다. 《기억기지역》에서 살펴본것처럼 이 서술자에는 pages_min 마당(언제나 확보하고있어야 하는 여유페지들의 수)와 pages_high 마당(더는 페지들을 회수할 필요가 없으므로 멈춰야 하는 안전수준)도 있다. 보통 pages_min 마당은 기억기구역의 크기를 페지단위로 나타낸 값을 128로 나눈 수고 pages_1ow는 pages_min의 두배이며 pages_high는 pages_min의 세배이다. 보통 kswapd_wait 대기렬에 kswapd 핵심부스레드를 삽입한다. 《페지들의 요청과 해제》에서 설명한것처럼 alloc_pages()함수가 pages_1ow보다 많은 페지들을 보유한 기억기지역을 찾는데 실패하면 첫번째 검사한 기억기지역의 need_balance 마당을 설정하고 kswapd를 깨운다.

kswapd 핵심부스레드는 kswapd()함수를 실행하여 매번 활성화될 때마다 다음과 같은 연산을 수행한다.

1. current(현재프로쎄스)의 상태를 TASK_RUNNING으로 설정하고 current를 kswapd_wait 대기렬에서 제거한다.

2. kswapd_balance()를 호출한다.(아래 참고)

3. tq_disk 작업대기렬에 대해 mn_task_queue()를 호출하여 블로크장치구동프로그람의 전략루틴을 활성화한다(《ll_rw_block()함수》 참고) 이렇게 하면 순서짜기된 입출력연산을 시작하여 결국 핵심부가 비동기완충기머리부와 페지캐쉬의 페지를 해제할 수 있도록 함으로써 기억기부담을 던다.

4. current의 상태를 TASK_INTERRUPTIBLE로 설정하고 current를 kswapd_wait 대기렬에 추가한다.

5. 모든 기억기지역서술자의 need_balance 기발을 검사한다.(《기억기지역》참고) 기발이 설정된것이 없으면 schedule()을 호출하여 kswapd 핵심부스레드를 잠들게 한다. 다시 실행되면 1단계로 이동한다.

kswapd_balance()함수는 모든 기억기지역의 need_balance 기발을 검사한다. 기발이 설정된 기억기지역에 대해 try_to_free_pages()를 호출하여 페지를 회수를 시작한다. try_to_free_pages()함수는 페지틀 SWAP_CLUSTER_MAX개를 회수하지 못할수도 있다. 이 경우에 한 프로쎄스를 제거한다. 제거하게 되면 kswapd_balance()는 자신을 1s동안 보류해서 그 동안 제거된 프로쎄스가 가지고있던 페지틀을 핵심부가 회수하도록 한다.

kswapd_balance()함수는 어떤 지역(또는 마디에 있는 다른 지역중 하나)의 여유페지틀수가 기억기구역서술자의 pages_high마당에 저장된 값보다 크게 될 때까지 기억기지역에 대해 try_to_free_pages()를 계속 호출한다.

제 5 장. 입출력장치

제 1 절. 새치기와 례외

일반적으로 새치기(interrupt)는 처리기가 실행하는 명령어의 순위를 바꾸는 사건이라고 정의한다. 이런 사건은 CPU내부와 외부에서 하드웨어적인 회로가 발생시키는 전기적인 신호에 해당한다.

새치기를 흔히 동기적(synchronous)인 새치기와 비동기적(asynchronous)인 새치기로 나눈다.

- 동기적인 새치기는 CPU조종장치가 명령을 실행하는 도중에 발생시키는데 조종장치는 명령어실행을 마친 후에만 이것을 발생시키기때문에 동기적이라고 한다.
- 비동기적인 새치기는 다른 하드웨어장치가 CPU박자신호과 관계없이 아무 때나 발생시킨다.

새치기는 간격시계(interval timer)와 입출력장치에 의해 발생한다. 례를 들어 사용자가 건반의 건을 누르면 새치기가 발생한다. 반면에 례외는 프로그람작성오유나 핵심부가 처리해야 하는 비정상적인 상황에 의해 발생한다. 프로그람작성오유인 경우 핵심부는 Unix프로그람작성자에게 익숙한 신호(signal)중 하나를 현재프로쎄스에 보내서 례외를 처리한다. 비정상적인 상황인 경우에는 페지절환(page fault)이나 (int명령을 통한)핵심부봉사요청 같은 비정상적인 상태를 복구하는데 필요한 모든 단계를 수행한다.

이 절에서는 모든 PC에서 공통적인 《전통적인》 새치기만을 다루며 일부구성방식에 있는 비표준새치기는 고찰하지 않는다. 례를 들어 무릎형콤퓨터(laptop)에서는 여기서 다루지 않는 종류의 새치기가 발생한다.

1. 새치기신호의 역할

이름 그대로 새치기신호는 프로쎄스가 정상적인 조종흐름밖의 코드로 방향을 바꾸는 방법을 제공한다. 새치기신호가 도착하면 CPU는 자기이 현재 수행하던 과제를 멈추고 새로운 과제로 전환해야 한다. CPU는 프로그람계수기(program counter)의 현재값(즉 eip와 cs등록기의 내용)을 핵심부방식탄창에 보관하고 발생한 새치기종류와 련관된 주소를 프로그람계수기에 넣어 수행한다.

여기서 핵심부가 한 프로쎄스를 다른 프로쎄스로 교체할 때 발생하는 문맥절환을 생각할수도 있을것이다. 그러나 새치기처리와 프로쎄스절환사이에는 중요한 차이가 있다. 새치기처리기나 례외처리기가 실행하는 코드는 프로쎄스가 아니며 새치기가 발생한 당시 현재프로쎄스의 문맥에서 실행되는 핵심부조종경로라고 할수 있다.(《례외처리기와 새치기처리기의 중첩실행》 참고)

새치기처리기는 핵심부조종경로이므로 프로쎄스보다 가볍다.(문맥이 더 적으며 시작하고 끝내는데 시간이 더 적게 든다.)

새치기처리는 다음과 같은 조건을 만족해야 하기때문에 핵심부가 수행하는 가장 민감한 과제중의 하나이다.

• 새치기는 핵심부가 빨리 끝마쳐야 하는 다른 과제를 하는 경우라도 상관없이 언제든지 발생할수 있다. 따라서 핵심부의 목표는 가능한 빨리 새치기에서 벗어나고 되도록 많은 과제를 나중으로 미루는것이다. 례를 들어 망선을 통해서 한개의 자료블로크가 도착하였다고 가정하자. 하드웨어가 새치기를 발생시켜 핵심부를 새치기하면 핵심부는 자료가 존재한다는 사실만 표시하고 처리기가 이전에 하던 일로 돌아가 과제를 끝마친 후 나머지 과제 (수신할 프로세스의 완충기로 자료를 복사한 후 프로세스를 다시 시작하는 등)를 처리한다. 이렇게 새치기가 발생할 때 핵심부가 이에 반응하여 할 일은 두가지로 나눌수 있다. 하나는 상반부(top half)로서 새치기가 발생할 때 핵심부가 곧바로 실행하는 일이고 다른 하나는 하반부(bottom half)로서 나중으로 미루는 일이다. 핵심부는 실행해야 할 하반부를 나타내는 모든 함수에 대한 지적자를 대기렬로 관리하다가 처리과정중 어떤 시점이 되면 대기렬에서 꺼내서 실행한다.

• 새치기는 언제든지 발생할수 있으므로 핵심부가 이미 새치기를 처리하고있을 때 다른 새치기(다른 종류의)가 발생할수 있다. 이것은 입출력장치의 운영을 최대화할수 있으므로 가능한 많이 허용해야 한다.(《례외처리기와 새치기처리기의 중첩실행》참고) 이런 리유로 새치기처리기는 해당 핵심부조종경로가 중첩되여도 실행할수 있도록 작성해야 한다. 마지막핵심부조종경로가 끝나면 핵심부는 새치기된 프로세스의 실행을 재개하거나 새치기신호로 인해 순서짜기가 다시 일어나면 다른 프로세스로 절환한다.

• 핵심부는 이전새치기를 처리하는 동안에도 새로운 새치기를 받을수 있지만 핵심부코드에는 새치기를 금지해야 하는 림계령역(critical region)이 존재한다. 이전조건에서 보는것처럼 핵심부는 (특히 새치기처리기는) 대부분의 시간을 새치기를 허용한 상태에서 보내야 하기때문에 이런 림계령역은 가능한 제한해야 한다.

2. 새치기와 례외

새치기와 례외를 다음과 같이 분류한다.

1) 새치기

o 마스크가능한 새치기

입출력장치가 제기하는 모든 새치기요청(IRQ)은 마스크가능한 새치기(maskable interrupt)를 발생시킨다. 마스크가능한 새치기는 마스크되거나 마스크되지 않은 상태중 하나에 있을수 있다.

조종장치는 마스크된 새치기를 마스크되여있는 동안 계속 무시한다.

o 마스크불가능한 새치기

일부 심각한 사건(하드웨어고장 같은)만이 마스크불가능한 새치기(nonmaskable interrup)를 발생시킨다. CPU는 마스크불가능한 새치기가 발생하면 이것을 항상 감시

한다.(intterupt.h참고)

2) 례외

(1) 처리기가 감시하는 례외

CPU가 명령을 실행하다가 비정상적인 상황을 발견할 때 발생한다. 이것은 CPU조종장치가 례외를 발생시킬 때 핵심부방식탄창에 보관하는 eip 등록기의 값에 따라 세 그룹으로 나눌수 있다.

o 장애

일반적으로 고칠수 있으며 일단 바로잡으면 프로그람은 재시작하여 이어서 실행할수 있다. 보관된 eip 값은 장애(fault)를 일으킨 명령의 주소이기때문에 례외처리기를 끝마치면 해당 명령으로 복귀할수 있다. 3장의 《페지절환례외처리기》에서 보았지만 처리기가 례외를 발생시킨 비정상적인 상태를 바로잡으려면 똑같은 명령으로 복귀하는 절차가 필요하다.

o 함정

함정(trap)을 발생시키는 명령을 실행하자마자 발생한다. 프로그람은 핵심부로부터 조종를 돌려받은 후에 이어서 실행할수 있다. 보관된 eip 값은 함정을 발생시킨 명령어 다음에 실행해야 하는 명령어의 주소이다. 함정은 완료한 명령을 다시 실행할 필요가 없을 때에만 일어난다. 함정의 기본용도는 오유수정이다. 이 경우 새치기신호는 오유수정기에 특정명령이 실행되였음을 알려주는 역할을 한다.(례를 들면 프로그람에 있는 중단점(breakpoint)에 도달) 사용자는 오유수정기가 제공하는 자료를 검사한 후 오유를 수정하고있는 프로그람이 다음명령으로부터 실행을 재개하도록 할수 있다.

o 중단

심각한 오유가 생겼을 때 발생한다. 조종장치가 어려운 상황에 빠져있고 eip 등록기에 례외를 일으킨 정확한 위치를 보관하지 못할수도 있다. 이것은 하드웨어고장이나 체계표에 잘못된 값이 들어있을 때 발생한다. 조종장치가 보낸 새치기신호는 조종권을 해당 중단(abort)례외처리기에 넘기려는 응답신호이다.이 처리기는 문제가 발생한 프로쎄스를 강제로 완료하는것외에 다른 선택권이 없다.

(2) 프로그람작성에 의한 례외

프로그람작성자의 요청에 의해 발생한다. 이것은 int나 int3 명령에 의해 발생한다. into(자리넘침(overflow)검사)와 bound(주소범위검사)명령도 검사하는 조건이 맞지 않은 경우 프로그람작성에 의한 례외를 발생시킨다. 조종장치는 이 례외를 함정으로 처리한다. 프로그람작성에 의한 례외를 종종 쏘프트웨어새치기(software interrupt)라고도 한다.이 례외의 일반적인 용도는 두가지로 정리할수 있다. 하나는 체계호출을 구성하는것이고 다른 하나는 오유수정기에 특별한 사건을 알려주는것이다.

각 새치기나 례외는 0부터 255까지의 수자로 구별한다. Intel은 이 8bit로 된 0이상의 수자를 벡토르(vector)라고 부른다. 마스크불가능한 새치기와 례외벡토르번호는 고정되여있다. 반면에 마스크가능한 새치기의 벡토르번호는 새치기조종기(Interrupt

controller)를 프로그람화하여 바꿀수 있다.

3. IRQ와 새치기

새치기요청(interrupt request)을 제기할수 있는 모든 하드웨어장치조종기는 IRQ 라는 출력선하나를 가진다.체계에 존재하는 모든 IRQ선은 새치기조종기(interrupt controller)라는 하드웨어회로의 입력단자로 련결된다. 새치기조종기가 하는 일은 다음과 같다.

1. IRQ선을 감시하고 신호가 발생하면 이것을 검사한다.
2. IRQ선에서 발생한 신호라면 다음과 같이 동작한다.

   a. 수신한 신호를 해당 벡토르로 변환한다.

   b. 새치기조종기의 입출력포구에 벡토르를 보관하여 CPU가 자료모선을 통해서 이것을 읽을수 있도록 한다.

   c. 발생한 신호를 처리기의 INTR단자로 보낸다. 즉 새치기를 발생시킨다.

   d. CPU가 프로그람가능한 새치기조종기(PIC: Programmable Interrupt Controller)의 입출력포구중 하나에 값을 써넣어 CPU가 새치기신호를 받았음을 알려줄 때까지 기다린다.이런 상태가 발생하면 INTR선을 0으로 만든다.

3. 1단계로 되돌아간다.

IRQ선은 0부터 시작해서 순차적으로 번호가 붙어있으며 첫번째 IRQ선을 보통 IRQ0으로 표시한다. Intel에서 IRQn에 부여하는 기본벡토르는 n+32이다. 앞에서 설명한대로 IRQ와 벡토르사이의 사영은 새치기조종기포구에 적절한 입출력명령을 보내서 수정할수 있다.

각 IRQ선을 선택적으로 금지할수 있으며 여러 IRQ를 금지하도록 PIC를 프로그람화할수도 있다. 즉 어떤 특정IRQ선에서 오는 새치기를 더는 발생시키지 않도록 PIC에 요청할수도 있고 그 반대로 새치기를 허용하게 할수도 있다. 금지된 새치기는 없어지는것이 아니며 PIC는 해당 새치기를 허용하자마자 새치기를 CPU에 전달한다. 이런 특성은 종류가 동일한 IRQ를 련속적으로 처리할수 있게 하기때문에 대부분의 새치기처리기에서 리용한다.

선택적으로 IRQ를 허용하거나 금지하는것과 별도로 마스크가능한 모든 새치기를 마스크하거나 마스크해제(unmask)할수도 있다. eflags등록기의 IF기발을 0으로 끄면 CPU는 일시적으로 PIC가 발생시키는 어떠한 마스크가능한 새치기도 무시한다. cli와 sti기호언어명령을 사용해서 각각 IF기발을 끄거나 켤수 있다. 다중처리기체계에서 새치기를 마스크하거나 마스크해제하는것은 각 CPU마다 자기만의 eflags등록기를 가지고있기때문에 좀 더 까다롭다. (irq.c, irq.h, irq_vectors.h 참고)

전통적으로 PIC는 8259A종류의 외부소자 두개를 중첩방식으로 련결하여 구성한다. 각 소자는 8개까지 서로 다른 새치기선을 처리할수 있다. 종속(slave)PIC의 INT 출력선은 주(master)PIC의 IRQ2에 련결되여있기때문에 사용할수 있는 IRQ의 개수는 15개로 제한된다.

4. 향상된 프로그람가능한 새치기조종기

앞에서는 단일처리기체계용으로 설계한 PIC를 설명하였다. 체계에 CPU가 하나밖에 없다면 주PIC의 출력선을 곧바로 CPU의 INTR단자로 련결하면 된다. 그렇지만 체계에 CPU가 두개이상 있다면 더는 이런식으로 할수 없고 좀 더 복잡한 PIC를 사용해야 한다.

SMP구조의 병렬성을 최대한 활용하려면 체계에 있는 모든 CPU로 새치기를 전달하는것이 필수적이다. 이런 리유로 Intel은 입출력 향상된 프로그람가능한 새치기조종기 (I/O APIC, I/O Advanced Programmable

Interrupt Controller)라는 구성요소를 도입해서 낡은 8259A프로그람가능한 새치기조종기를 대체하였다. 나가서 현재 출하되는 모든 Intel CPU에는 국부 APIC(local APIC)가 들어있다. 각 국부 APIC는 32bit 등록기와 내부박자, 국부시계장치, 국부새치기용으로 예약된 두개의 부가적인 IRQ선인 LINT0과 LINT1을 가진다. CPU내부에 있는 모든 국부 APIC를 외부에 있는 입출력APIC로 련결하여 다중APIC체계를 구성한다.(io_apic.c, mach_apic.h 참고)

그림 5-1은 다중 APIC체계의 구조를 도식적으로 보여준다. APIC모선(APIC bus)은 앞단에 있는 입출력APIC와 국부APIC를 련결한다. 장치로부터 나온 IRQ선은 입출력 APIC로 련결되여 입출력 APIC는 국부APIC의 립장에서 보면 경로기(router)역할을 한다. APIC모선은 펜티움3이나 그 이전 처리기를 탑재한 기판에서는 직렬로 된 선 3개로 이루어진다. 펜티움4부터는 체계모선를 리용하여 APIC모선를 구성한다. 그렇지만 APIC모선과 APIC모선통보는 쏘프트웨어에 보이지 않기때문에 더는 설명하지 않는다.

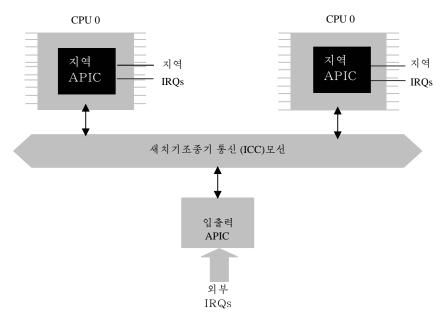



그림 5-1. 다중 APIC 체계

입출력 APIC는 IRQ선 24개와 새치기재지정표(interrupt redirection table)입구점 24개, 프로그람가능한 등록기, APIC모선를 통하여 APIC통보를 보내고받을수 있는 통보장치(message unit)로 구성된다. 8259A의 IRQ단자와는 달리 단자번호와 새치기우선순위는 이무런 관계가 없다.재지정표에 있는 입구점을 프로그람화하여 개별적으로 새치기벡토르와 우선순위, 목적지처리기, 처리기선택방법을 지정할수 있다. 재지정표에 있는 정보를 사용하여 각 외부IRQ신호를 APIC모선를 통해 국부APIC장치 하나이상으로 전달하는 통보로 변환한다.

외부하트웨어장치에서 오는 새치기요청을 사용가능한 CPU사이에서 분배하는 방법은 두가지이다.

o **정적분배**

IRQ 신호를 해당 재지정표입구점에서서 지정하고있는 국부APIC로 전달한다. 한번에 지정한 CPU 하나 혹은 CPU의 부분집합, 모든 CPU로(방송방식) 새치기를 전달한다.

o **동적분배**

새치기신호를 우선순위가 가장 낮은 프로쎄스를 실행하는 처리기의 국부APIC로 전달한다. 모든 국부APIC는 프로그람가능한 과제우선순위등록기(TPR, Task Priority Register)를 가지고있어 현재 실행중인 프로쎄스의 우선순위를 계산할 때 사용한다. Intel에서는 프로쎄스절환을 할 때마다 조작체계핵심부가 이 등록기를 바꿀것이라고 예상한다.

같은 가장 낮은우선순위를 가진 CPU가 두개이상이라면 중재

(arbitration)기법을 사용하여 CPU사이에서 부하를 분배한다. 국부APIC에 있는 중재우선순위등록기(arbitration priority register)를 리용하여 각 CPU마다 0에서 15사이에 있는 중재우선순위를 지정한다. 각 국부APIC는 서로 고유한값을 가진다.

새치기를 CPU에 전달할 때마다 해당 CPU의 중재우선순위를 자동으로 설정하고 다른 모든 CPU에 있는 중재우선순위를 1씩 증가시킨다. 중재우선순위등록기값이 15보다 커지면 이 값을 바로 전에 새치기를 전달한 CPU가 가지고있던 중재우선순위값에 1을 증가시킨값으로 설정한다. 따라서 과제우선순위가 같은 CPU사이에서 원형(round-robin)방식으로 새치기를 분배하게 된다.

다중 APIC체계는 처리기사이에서 새치기를 분배하는것외에도 CPU가 처리기간 새치기(interprocessor interrupt)를 발생시킬수 있게 한다. CPU는 다른 CPU로 새치기를 보내고 싶을 때 자기의 국부APIC에 있는 새치기명령등록기(ICR, Interrupt command Register)에 새치기벡토르와 대상CPU에 있는 국부APIC의 식별자를 기록한다. 그러면 APIC모선를 통해 목적지인 국부APIC로 통보가 전달되여 그 CPU에서 해당하는 새치기가 발생한다.

처리기간새치기(간단히 줄여서 IPI라 한다.)는 SMP구성방식의 일부로서 Linux는 CPU사이에서 통보를 교환할 때 이것을 활발히 사용한다.(뒤에 나오는 《새치기봉사루

틴》참고) 지금 나오는 대부분의 단일처리기체계는 입출력APIC소자을 내장하고있으며 이 소자를 서로 다른 두가지 방식으로 설정할수 있다.

- CPU에 련결된 표준8259A류형의 외부 PIC로 설정한다. 국부APIC를 끄고 두개의 국부IRQ선인 LINT0과 LINT1을 각각 INTR와 NMI단자로 설정한다.
- 표준입출력APIC로 설정한다. 국부APIC를 켜고 입출력APIC를 통해 외부새치기를 수신한다.

5. 례외

80x86극소형처리기는 대략 20여개의 서로 다른 례외를 발생시킨다. 핵심부는 각 례외류형에 따라 전용례외처리기를 제공해야 한다. 일부 례외의 경우 CPU조종장치는 하드웨어오유코드(hardware error code)를 만들어 례외처리기를 시작하기 전에 이것을 핵심부방식탄창에 넣기도 한다.

다음목록은 80x86처리기에 있는 례외의 벡토르와 이름, 류형 그리고 간단한 설명이다.

0 -《나누기오유(divide error)》(장애)

프로그람이 어떤 정수값을 0으로 나누려고 할 때 발생한다.

1 -《오유수정(Debug)》(함정 또는 장애)

eflags의 T기발을 설정(오유수정하려는 프로그람을 행단위로 실행할 때 매우 쓸모있다.)한 때나 명령주소나 피연산자가 활성화된 오유수정등록기에서 지정한 범위안에 들어갈 때 발생한다.

2 -사용하지 않음

마스크불가능한 새치기(NMI단자를 사용하는 새치기)용으로 예약하고있다.

3 -《중단점(Breakpoint)》(함정)

Int3(중단점)명령어(보통 오유수정기가 삽입한다.)에 의해 발생한다.

4 -《자리넘침(Overflow)》(함정)

into(자리넘침검사)명령을 실행할 때 eflags의 OF(자리넘침)기발이 설정되여있는 경우에 발생한다.

5 -《범위검사(Bound check)》(장애)

bound(주소범위검사)명령을 실행할 때 지정한 피연산자가 유효한 주소범위를 벗어나있는 경우에 발생한다.

6 -《잘못된 연산코드(Invalid opcode)》(장애)

CPU실행장치(execution unit)가 잘못된 연산코드(opcode, 수행할 연산을 결정하는 기계어명령의 일부)를 발견한 경우에 발생한다.

7 -《장치를 사용할수 없음(Device not available)》(장애)

cr0의 TS기발을 설정한 상태에서 확장명령(escape instrucion)이나 MMX, XMM 명령을 실행한 경우이다.

8 - 《이중장애(Double fault)》(중단)

CPU가 이전에 발생한 례외를 처리하는 처리기를 호출하는 과정에서 례외가 다시 발생한 때이다. 보통은 례외 두개를 련속해서 처리할수 있지만 종종 련속적으로 처리할 수 없는 경우가 있는데 바로 이때 이 례외가 발생한다. (doublefault.c 참고)

9 - 《보조처리기토막초과(Coprocessor segment overrun)》(중단)

외부수값보조처리기에 문제가 발생한 때이다.(이전 80386극소형처리기에만 해당한다.)

10 -《잘못된 TSS(Invalid TSS)》(장애)

CPU가 잘못된 과제상태토막를 가진 프로쎄스로 문맥절환하려 할 때 발생한다.

11 –《존재하지 않는 토막(Segment not present)》(장애)

기억기에 존재하지 않는 토막(토막서술자의 Segment-Present 기발이 0인 토막)을 참조하는 경우에 발생한다.

12 -《탄창토막(Stack segment)》(장애)

명령이 탄창토막의 제한을 넘어서려고 했거나 ss가 가리키는 토막이 기억기에 존재하지 않는 경우에 발생한다.

13 –《일반보호(Generalprotection)》(장애)

80x86의 보호방식에 있는 보호규칙중 하나를 위반한 경우에 발생한다.

14 -《페지절환(Page Fault)》(장애)

주소로 참조한 페지가 기억기에 존재하지 않거나 해당 페지표입구점이 비여있거나 페지화보호수법을 위반한 경우이다.

15 - 예약

16 -《류동소수점오유(Floating point error)》(장애)

CPU에 통합되여 들어있는 류동소수점장치에서 수자자리넘침이나 0으로 나누기 같은 오유상황이 발생했음을 알려주는 경우이다.

17 -《정렬검사(Alignment check)》(장애)

피연산자의 주소가 옳바로 정렬되여있지 않은 경우이다.(례를 들어 long 자료형 옹근수의 주소가 4의 배수가 아닌 경우)

18 -《기계검사(Machine check)》(중단)

기계검사수법이 CPU나 모선오유를 발견한 경우이다.

19 -《SIMD류동소수점》(중단)

CPU소자에 들어있는 SSE나 SSE2장치에서 류동소수점연산을 할 때 오유상태가 발생하여 이것을 알려준 경우이다.

Intel은 20에서 31까지 범위를 나중에 개발용으로 사용하기 위해 예약해 두었다. 표 5-1에서 보는바와 같이 각 례외마다 특정례외처리기가 있어 례외를 처리한다.(뒤에 나오는 《례외처리》참고) 이 례외처리기는 대개 례외를 발생시킨 프로쎄스에 Unix신

호를 보낸다.

표 5-1. 례외처리기가 보내는 신호

№	례　외	례외조종기	신 호
0	나누기오유	Divide_error()	SIGFPE
1	오유수정	Debug()	SIGTRAP
2	NMI	Nmi()	없음
3	중단점	Int3()	SIGTRAP
4	자리넘침	overflow()	SIGSEGV
5	범위검사	bounds()	SIGSEGV
6	잘못된 연산코드	invalid_op()	SIGILL
7	장치를 사용할수 없음	device_not_available()	SIGSEGV
8	이중실패	double_fault()	SIGSEGV
9	보조프로쎄스토막넘침	coprocessor_segment_overrun()	SIGFPE
10	잘못된 TSS	invalid_tss()	SIGSEGV
11	존재하지 않는 토막	segment_not_present()	SIGBUS
12	탄창토막	Stack_segment()	SIGBUS
13	일반보호	general_protection()	SIGSEGV
14	페지실패	Page_fault()	SIGSEGV
15	예약	없음	없음
16	류동소수점오유	coprocessor_error()	SIGFPE
17	정렬검사	alignment_check()	SIGBUS
18	기계검사	machine_check()	없음
19	SIMD류동소수점	simd_coprocessor_error()	SIGFPE

6. 새치기서술자표

새치기서술자표(IDT: Interrupt Descriptor Table)라는 체계표는 각 새치기와 새치기처리기의 주소, 례외벡토르와 례외처리기의 주소를 련결한다. 핵심부는 새치기를 허용하기 전에 IDT를 옳바르게 초기화해야 한다. IDT 형식은 4장에서 살펴본 GDT와 LDT의 형식과 비슷하다. 각 입구점는 8B크기의 서술자로 되여있으며 새치기벡토르나 례외벡토르 하나에 대응한다. 따라서 IDT를 보관하는데는 최대 256×8=2048B가 필요하다. IDT의 기정입구점수는 256으로 제정되여있다.

idtr CPU등록기가 있어 IDT를 기억기의 어느 위치에나 둘수 있다. idtr등록기에

는 IDT의 기본물리주소와 범위(최대길이)가 들어간다. 새치기를 허용하기 전에 lidt아쎔블리어명령을 사용하여 idtr를 초기화해야 한다. IDT에는 세 종류의 서술자가 들어갈수 있다. 그림 5-2에 각 서술자에 들어가는 64bit의 의미를 보여준다. 이중 40~43bit에 있는 Type마당이 서술자의 종류를 구별한다.

세 종류의 서술자는 다음과 같다.

과제문

새치기신호가 발생할 때 현재프로쎄스를 대체할 프로쎄스의 TSS선택기를 가진다. Linux에서는 과제문(task gate)을 사용하지 않는다.

새치기문

새치기나 례외처리기의 토막선택기와 토막내편위를 가진다. 해당 토막으로 조종를 넘길 때 처리기는 IF기발을 0으로 설정하여 마스크가능한 새치기가 더는 발생하지 않게 한다.

함정문

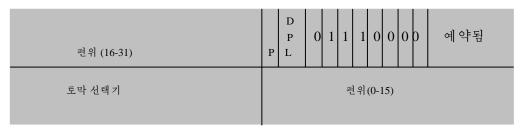
새치기문(interrupt gate)과 비슷하지만 해당 토막으로 조종를 넘길 때 처리기가 IF기발을 바꾸지 않는다는 차이가 있다. 앞으로 《새치기와 함정, 체계문》에서 살펴보지만 Linux는 새치기를 처리할 때에는 새치기문, 례외를 처리할 때에는 함정문을 사용한다.

과제문서술자

63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32

예약됨	P	DPL	0	0	1	0	1	예약됨
TSS 토막 선택기	예약됨							

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

새치기문서술자

함정문서술자

63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32

편위(16-3)	P	DPL	0	1	1	1	1	0	0	0	예약됨
토막 선택기	편위(0-15)										

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

그림 5-2. 문서술자형식

7. 새치기와 례외의 하드웨어적인 처리

이제 CPU조종장치가 어떻게 새치기와 례외를 처리하는지 살펴보자. 여기서는 핵심부초기화가 이미 끝나서 CPU가 보호방식에서 동작하고있다고 가정한다. 어떤 명령을 실행한 후 cs와 eip등록기쌍은 다음에 실행할 명령어의 론리주소를 가진다. 다음명령을 처리하기 전에 조종장치는 이전 명령을 실행하는 동안 새치기나 례외가 발생했는지 검사한다. 만약 둘중 하나라도 발생하였다면 조종장치는 다음과 같이 동작한다.

1. 발생한 새치기나 례외에 해당하는 벡토르 i(0≦I≦255)를 알아낸다.

2.idtr등록기가 가리키는 IDT에서 i번째 입구점을 읽어들인다.(입구점에 새치기문이나 함정문이 들어 있다고 가정한다.)

3. gdtr등록기에서 GDT의 기본주소를 가져와서 IDT입구점에 있는 선택기가 가리키는 토막서술자를 GDT에서 읽어들인다. 이 서술자는 새치기처리기나 례외처리기를 포함한 토막의 시작주소를 지정한다.

4. 새치기가 인증된 곳에서 발생한것인지 검사한다. 먼저 cs등록기의 마지막 두 비트에 들어있는 현재특권준위(CPL, Current Privilege Level)와 GDT에 들어있는 토막서술자의 서술자특권준위(DPL Descriptor Privilege Level)를 비교한다. 새치기처리기는 새치기를 발생시킨 프로그람보다 낮은 특권을 가질수 없으므로 CPL이 DPL보다 낮으면 일반보호(general protection)례외를 발생시킨다.

프로그람작성에 의한 례외(Programmed exception)의 경우 보안검사를 추가로 한다. CPL과 IDT에 들어있는 문(gate)서술자의 DPL을 비교하여 DPL이 CPL보다 낮으면 일반보호례외를 발생시킨다. 이 마지막검사는 사용자응용프로그람이 특정함정문이나 새치기문에 접근하는것을 막을수 있게 한다.

5. 특권준위의 변화가 일어날것인지 즉 선택한 토막서술자의 DPL과 CPL이 다른

지 검사한다. 다르다면 조종장치는 새로운 특권준위에 맞는 탄창을 사용하기 시작해야 한다.

다음단계로 이것을 수행한다 .

a.current 프로쎄스의 TSS토막에 접근하려고 tr등록기를 읽어들인다.

b.새로운 특권준위과 관련된 옳바른 탄창토막과 탄창지적자의 값으로 ss와 esp등록기를 설정한다. 이 값은 TSS에 들어있다.

c. 이전특권준위과 련관된 탄창의 론리주소 ss와 esp의 이전값을 새로운 탄창에 보관한다.

6. 장애(fault)가 발생하였다면 례외를 발생시킨 명령의 론리주소로 cs와 eip를 설정하여 이 명령을 다시 실행할수 있게 한다.

7. eflags와 cs, eip의 내용을 탄창에 보관한다.

8. 례외에 따른 하드웨어오유코드가 있으면 이것을 탄창에 보관한다.

9. cs와 eip를 각각 IDT의 i번째 입구점에 보관된 문(gate)서술자의 토막선택기와 편위마당으로 설정한다. 이 값은 새치기처리기나 례외처리기의 첫번째 명령의 론리주소이다. 조종장치는 마지막 단계로 새치기처리기나 례외처리기로 이행한다. 다시 말하면 새치기신호를 처리한 후에 조종장치가 처음으로 실행하는 명령은 선택한 처리기의 첫 명령어가 된다,

새치기나 례외를 처리하고 나면 해당 처리기는 iret명령을 호출하여 새치기되였던 프로쎄스로 조종권을 돌려주어야 한다. iret명령을 실행하면 조종장치는 다음과 같이 동작한다.

1. 탄창에 보관했던 값으로 cs와 eip, eflags등록기를 복구한다. 탄창에서 eip다음에 하드웨어오유코드가 들어있다면 iret를 실행하기 전에 이것을 꺼내야 한다.

2. 처리기의 CPL이 cs의 아래자리 두 비트에 들어있는 값과 같은지 검사한다.(같다면 새치기된 프로쎄스가 처리기와 똑같은 특권준위에서 실행중이였다는 사실을 의미한다.) 값이 같으면 lret는 실행을 완료하고 그렇지 않으면 다음단계로 진행한다.

3. 탄창에서 ss와 esp등록기를 가져와서 이전특권준위과 련관된 탄창으로 돌아간다.

4. ds와 es, fs, gs토막등록기의 값을 검사한다. 이중 하나라도 DPL이 CPL보다 낮은 토막서술자를 가리키면 해당 토막등록기를 지운다. 조종장치가 이런 일을 하는 리유는 CPL이 3인 사용자방식프로그람이 이전에 핵심부코드에서 사용한 DPL이 0인 토막등록기를 사용하는것을 막기 위해서이다. 이 등록기를 지우지 않으면 악의적인 사용자방식프로그람이 이것을 리용하여 핵심부주소공간에 접근할수 있다.

8. 례외처리기와 새치기처리기의 중첩실행

핵심부는 새치기나 례외를 처리할 때 별개의 련속된 명령인 핵심부조종경로(kernel controll path)를 새로 시작한다. 례를 들어 프로쎄스가 체계호출을 진행하면 해당 핵

심부조종경로의 첫번째 명령은 등록기의 내용을 핵심부방식탄창에 보관하고 마지막 명령은 등록기의 내용을 복구하고 CPU를 사용자방식으로 돌려보낸다. Linux는 CPU가 새치기를 다루는 핵심부조종경로를 실행할 때에는 프로쎄스절환이 일어나지 않도록 설계되였다. 그러나 핵심부조종경로는 얼마든지 중첩할수 있다. 새치기처리기는 다른 새치기처리기를 가로챌수 있어 결과적으로 핵심부조종경로를 중첩해서 실행하게 된다. 다시한번 강조하지만 핵심부가 핵심부조종경로를 중첩해서 다루는 동안에 현재프로쎄스는 바뀌지 않는다. 핵심부에 오유가 없다고 가정하면 대부분의 례외는 CPU가 사용자방식에 있을때 발생한다. 실제로 례외는 프로그람작성오유나 오유수정기에 의해 발생한다.그러나 핵심부방식에서도 페지절환(page fault)례외가 발생할수 있다. 이것은 프로쎄스가 자기의 주소공간에는 들어가지만 현재주기억에 존재하지 않는 페지의 주소를 지정한 경우에 일어난다. 이런 례외를 처리하는 동안 핵심부는 현재프로쎄스를 보류하고 요청한 페지를 사용할수 있을 때까지 다른 프로쎄스로 절환할수 있다. 페지절환례외를 처리하는 핵심부조종경로는 프로쎄스가 처리기를 다시 할당받자마자 실행을 재개한다. 페지절환례외처리기는 더는 례외를 발생시키지 않으므로 례외와 관련하여 조종경로는 최대 2개까지 중첩될수 있다.(첫번째는 체계호출을 진행할 때 두번째는 페지절환에 의해 발생한다.)

례외와는 대조적으로 입출력장치가 발생시키는 새치기는 이것을 처리하는 핵심부조종경로가 현재프로쎄스를 대신해서 동작하더라도 현재프로쎄스와 관련된 자료구조를 참조하지 않는다. 실제로 새치기가 발생할 때 현재 어떤 프로쎄스가 실행중인지 예측하는 것은 불가능하다.

새치기처리기는 다른 새치기처리기와 례외처리기를 선취할수 있다. 반대로 례외처리기는 새치기처리기를 선취할수 없다. 핵심부방식에서 일어날수 있는 유일한 례외는 방금전에 설명한 페지절환이다. 그러나 새치기처리기는 잠재적으로 프로쎄스절환을 일으키는 가능성을 가진 페지절환을 발생시킬수 있는 어떠한 동작도 하지 않는다. Linux는 다음과 같은 두가지 주요리유때문에 핵심부조종경로의 중첩을 허용한다.

· 프로그람가능한 새치기조종기와 장치조종기의 처리량을 늘이기 위해서이다. 실례를 들어 장치조종기가 IRQ선으로 새치기신호를 발생시켰다고 가정해보자. PIC는 이것을 외부새치기로 변환하고 PIC와 장치조종기는 PIC가 CPU로부터 응답을 받을 때까지 차단된 상태로 남는다. 핵심부조종경로를 중첩함으로써 핵심부는 이전에 발생한 새치기를 처리하는 도중이라도 응답신호를 보낼수 있다.

·우선순위가 없는 새치기모형을 구성하기 위해서이다. 새치기처리기는 다른 새치기를 지연할수 있으므로 하드웨어장치사이에 미리 우선순위를 정할 필요가 없다. 이것은 핵심부코드를 간단하게 하고 호환성을 높여준다.

다중처리기체계에서는 여러 핵심부조종경로를 동시에 실행할수 있다. 나가서 례외를 처리하는 핵심부조종경로는 한 CPU에서 실행되다가 프로쎄스절환이 일어난 후 다른 CPU 로 옮겨져 실행될수도 있다.

9. 새치기서술자표 초기화

이제 Intel처리기가 하드웨어준위에서 새치기와 례외를 어떻게 다루는지 리해하였을것이다. 여기서는 새치기서술자표 (IDT)을 어떻게 초기화하는지 살펴보자.

앞에서 핵심부는 새치기를 허용하기 전에 IDT표의 시작주소를 idtr등록기에 보관하고 이 표의 모든 입구점을 초기화해야 한다고 설명하였다. 이런 과제는 모두 체계를 초기화하는 동안 이루어진다.

사용자방식프로쎄스는 int명령을 사용하여 0에서 255사이의 임의의 벡토르에 해당하는 새치기신호를 발생시킬수 있다. 따라서 IDT를 초기화할 때에는 사용자방식프로쎄스가 int명령으로 모방한 불법새치기와 례외를 막을수 있도록 주의해야 한다. 이것은 새치기나 함정문서술자의 DPL마당을 0으로 설정하여 해결할수 있다. 프로쎄스가 이런 식으로 새치기신호를 발생시키려고 하면 조종장치 CPL값과 DPL마당을 비교하여 일반보호례외를 발생시킨다.

그러나 사용자방식프로쎄스가 프로그람작성에 의한 례외를 발생시켜야 하는 경우도 있다. 이것은 해당 새치기나 함정문서술자의 DPL마당을 3으로 즉 가장 높은 값으로 설정하여 허용할수 있다.

이제 Linux에서 이런 방책을 어떻게 구성하는지 보자.

1) 새치기와 함정, 체계문

앞서 《새치기서술자표》에서 설명한것처럼 Intel은 과제와 새치기, 함정문서술자라는 3가지 새치기서술자를 제공한다. 과제문서술자는 Linux와 관계없지만 새치기와 함정문서술자는 새치기서술자표에서 많이 사용된다. Linux에서는 Intel과는 조금 다른 분류법과 용어를 사용하여 이것을 다음과 같이 분류한다.

새치기문

사용자방식프로쎄스에서 접근할수 없는(DPL마당이 0인) Intel의 새치기문.

모든 Linux새치기처리기는 새치기문(interrupt gate)을 통해서만 실행할수 있으며 모두 핵심부방식으로 제한한다.

체계문

사용자방식프로쎄스에서 접근할수 있는(DPL마당이 3인) Intel의 함정문.

벡토르 3, 4, 5, 128에 해당하는 례외처리기 4개를 체계문(system gate)을 통해 실행한다. 따라서 사용자방식에서는 int3, into, bound, int 0x80 이렇게 4개의 명령을 사용할수 있다.

함정문

사용자방식프로쎄스에서 접근할수 없는(DPL마당이 0인) Intel의 함정문.

Linux는 대부분의 례외처리기를 함정문(trap gate)을 통해 실행한다.(traps.c 참고)

IDT에 문을 넣을 때에는 다음과 같은 구성방식의존적인 함수를 사용한다.

set_intr_gate(n, addr)

n번째 IDT입구점에 새치기문을 넣는다. 문에 있는 토막선택기는 핵심부코드의 토막선택기로, 편위마당은 새치기처리기주소인 addr로 설정한다. DPL마당은 0으로 설정한다.

set_system_gate(n, addr)

n번째 IDT입구점에 함정문을 넣는다. 문에 있는 토막선택기는 핵심부코드의 토막선택기로, 편위마당은 례외처리기주소인 addr로 설정한다. DPL마당은 3으로 설정한다.

set_trap_gate(n, addr)

앞의 함수와 비슷하지만 DPL을 0으로 설정한다.

이 함수들은 _set_gate()를 호출하여 대면의 공통성을 실현한다.

```
#define _set_gate(gate_addr,type,dpl,addr,seg)  \
do {
   \
  int __d0, __d1;                                                  \
  __asm__ __volatile__ ("movw %%dx,%%ax\n\t"  \
   "movw %4,%%dx\n\t"                                              \
   "movl %%eax,%0\n\t"                                        \
   "movl %%edx,%1"                                            \
   :"=m" (*((long *) (gate_addr))),                      \
    "=m" (*(1+(long *) (gate_addr))), "=&a" (__d0), "=&d" (__d1) \
   :"i" ((short) (0x8000+(dpl<<13)+(type<<8))),\
    "3" ((char *) (addr)),"2" ((seg) << 16));    \
} while (0)
```

2) 림시적인 IDT초기화

콤퓨터가 실방식(real mode)에서 동작하고있을 때 BIOS는 IDT를 초기화하고 사용한다. Linux에서는 BIOS함수를 사용하지 않기때문에 일단 Linux가 체계를 장악한 후에 IDT를 주기억의 다른 령역으로 옮기고 두번째 초기화를 수행한다. 입구점이 256개인 idt_table표에 IDT를 보관하고 idt변수로 이 IDT를 가리킨다. 6B크기의 idt_descr 변수에 IDT의 크기와 주소를 보관한다. 이 변수는 핵심부가 lidt기호언어명령을 사용해서 idtr등록기를 초기화할 때에만 사용된다.

핵심부초기화과정에서 setup_idt() 기호언어함수를 호출해서 idt_table의 입구점 256개를 모두 ignore_int() 새치기처리기를 가리키는 똑같은 새치기문으로 채운다. 일부펜티움모형에서는 사용자방식프로그람이 체계를 멈추게 할수 있는 foo라는 오유가 있

다. Linux는 이런 CPU에서 동작할 때 IDT를 쓰기금지된 페지들에 보관하여 이 오유를 막는다. 사용자는 핵심부를 콤파일할 때 이것을 사용할것인지를 항목으로 선택할수 있다.(head.S 참고)

```
setup_idt:
lea ignore_int, %edx
movl $ (_KERNEL_CS << 16) , %eax
movw %dx, %ax /* 분구 =0x0010= cs */
movw $0x8e00, %dx /* 새치기 문, DPL=0, 기억기에 존재 */
lea idt_table, %edi
mov $256, %ecx
rp_sidt:
movl %eax, (%edi)
movl %edx, 4 (%edi)
addl $8, %edi
dec %ecx
jne rp_sidt
ret
```

기호언어로 작성한 ignore_int()새치기처리기는 다음과 같은 동작을 수행하는 빈운전기(null handler)라고 볼수 있다.

1. 일부등록기의 내용을 탄창에 보관한다.
2. printk() 함수를 호출해서 《Unknown interrupt》체계통보를 출력한다.
3. 등록기를 탄창에 있는 등록기내용으로 복구한다.
4. iret 명령을 실행하여 새치기된 프로그람으로 돌아간다.

이 처리의 본체를 보면 다음과 같다.

```
ignore_int:
   cld
   pushl %eax
   pushl %ecx
   pushl %edx
   pushl %es
   pushl %ds
   movl $(__KERNEL_DS),%eax
   movl %eax,%ds
   movl %eax,%es
   pushl 16(%esp)
```

```
pushl 24(%esp)
pushl 32(%esp)
pushl 40(%esp)
pushl $int_msg
call printk
addl $(5*4),%esp
popl %ds
popl %es
popl %edx
popl %ecx
popl %eax
iret
```

ignore_int() 처리기는 절대로 실행되여서는 안된다. 작업대나 일지파일에 《Unknown interrupt》 통보가 나오면 하드웨어적인 문제가 있거나(입출력장치에서 예상치 못한 새치기발생) 핵심부에 문제가 있음(새치기나 례외를 제대로 처리하지 못하고있음)을 의미한다. 이와 같은 림시적인 초기화후에 핵심부는 IDT에 있는 빈 운전기중 일부를 의미 있는 함정처리기와 새치기처리기로 교체하는 두번째 단계를 밟는다. 이 작업을 하고 나면 IDT는 조종장치가 발생시키는 서로 다른 례외와 PIC가 인식한 각 IRQ에 대해서 전용새치기나 함정, 체계문을 가지게 된다.

아래에서는 례외와 새치기를 각각 어떻게 처리하는지 자세히 설명한다.

10. 례외처리

Linux는 CPU가 제기하는 대부분의 례외를 오유상태로 해석한다. 이중 하나라도 발생하면 핵심부는 례외를 일으킨 프로쎄스에 신호를 보내서 비정상적인 상태가 발생했음을 알린다. 례를 들어 프로쎄스에서 0으로 나누기를 하면 CPU는 《나누기 오유》 례외를 발생시키고 해당 례외처리기는 SIGFPE 신호를 현재프로쎄스에 보내서 복구하는데 필요한 일을 하거나 (이 신호용으로 따로 신호처리기를 설정하지 않은 경우) 프로쎄스를 중단한다.

그런데 Linux에서 하드웨어자원을 더 효률적으로 관리하려고 CPU례외를 활용하는 경우가 몇가지 있다. cr0등록기의 TS기발과 《장치를 사용할수 없음》례외를 같이 사용하여 핵심부가 CPU의 류동소수점등록기에 새로운 값을 적재하게 할수 있다. 두번째 경우는 프로쎄스에 새로 페지틀을 할당하는 작업을 가능한 순간까지 미룰 때 사용하는 페지절환례외이다.이 례외는 오유일수도 있고 아닐수도 있기때문에 이 처리기는 복잡하다.

례외처리기는 다음 세 부분으로 이루어진 표준구조를 가진다.

1. 핵심부방식탄창에 대부분의 등록기내용을 보관한다.(이 부분은 아셈블러로 작성되여있다.)

2. 고수준 c함수에서 례외를 처리한다.

3. ret_from_exception() 함수를 호출해서 처리기에서 빠져나온다.

례외를 활용하려면 인식한 각 례외를 위한 례외처리기함수로 IDT를 옳바로 초기화해야 한다. 마스크할수 없는 새치기와 례외에 관한 모든 IDT입구점에 최종적인 값, 즉 례외를 처리하는 함수를 지정하는것이 trap_init()함수의 역할이다. 이것을 위해 set_trap_gate와 set_intr_gate, set_system_gate 마크로를 사용한다.

```
    set_trap_gate(0,&divide_error);
    set_intr_gate(1,&debug);
    set_intr_gate(2,&nmi);
    set_system_gate(3,&int3); /* int3-5 can be called from all */
    set_system_gate(4,&overflow);
    set_system_gate(5,&bounds);
    set_trap_gate(6,&invalid_op);
    set_trap_gate(7,&device_not_available);
    set_task_gate(8,GDT_ENTRY_DOUBLEFAULT_TSS);
    set_trap_gate(9,&coprocessor_segment_overrun);
    set_trap_gate(10,&invalid_TSS);
    set_trap_gate(11,&segment_not_present);
    set_trap_gate(12,&stack_segment);
    set_trap_gate(13,&general_protection);
    set_intr_gate(14,&page_fault);
    set_trap_gate(15,&spurious_interrupt_bug);
    set_trap_gate(16,&coprocessor_error);
    set_trap_gate(17,&alignment_check);
#ifdef CONFIG_X86_MCE
    set_trap_gate(18,&machine_check);
#endif
    set_trap_gate(19,&simd_coprocessor_error);
    set_system_gate(SYSCALL_VECTOR,&system_call);
    set_call_gate(&default_ldt[0],lcall7);
    set_call_gate(&default_ldt[4],lcall27);
```

이제 례외처리기가 호출될 때 어떤 일이 일어나는지 전형적인 처리기를 통해 알아보자.

1) **레외처리기를 위한 등록기보관**

일반레외처리기의 이름을 handler_name이라고 하자.(모든 레외처리기의 실제 이름은 방금의 코드에 나타나있다.) 각 레외처리기는 다음 기호명령으로 시작한다.

```
handler_name:
pushl $0 /* 일부레외에서만 */
pushl $do_handler_name
jmp_error_code
```

레외가 발생할 때 조종장치가 자동으로 탄창에 하드웨어오유코드를 넣지 않으면 해당 기호언어코드는 push $0명령을 추가하여 탄창에 빈값을 넣는다. 다음으로 고수준 C 함수의 주소를 탄창에 넣는다. 이 함수이름은 레외처리기이름앞에 do_라는 앞붙이를 불인것이다.

error_code라는 표가 붙은 기호언어코드토막은 《장치를 사용할수 없음》레외를 제외한 모든 레외처리기에서 똑같이 사용한다.

이 코드는 다음과 같은 단계를 밟는다.

1.고수준C함수에서 사용할수도 있는 등록기를 탄창에 보관한다.

2.cld명령을 실행하여 eflags등록기의 방향기발(direction flag) DF를 지운다. 이렇게 하면 문자렬명령(73)에서 edi와 csi가 자동증가한다.

3. 탄창의 esp + 36위치에 보관되여있는 하드웨어오유코드를 eax에 보관하고 이 탄창위치의 값을 -1로 설정한다. 0x80레외를 다른 레외와 구별할 때 이 값을 사용한다.

4. edi등록기를 탄창의 esp+32 위치에 보관되여있는 고수준 do_handler_name() C함수의 주소로 설정하고 es등록기의 내용을 이 탄창위치에 보관한다.

5. ds와 es등록기를 핵심부자료토막선택기로 설정하고 ebx등록기를 현재프로쎄스 서술자의 주소로 설정한다

6.고수준C함수에 전달할 파라메터 즉 레외의 하드웨어오유코드와 탄창에서 사용자방식등록기를 보관한 곳의 주소를 탄창에 보관한다.

7.edi에 보관한 고수준C함수의 주소로 함수호출을 한다.

마지막단계를 실행하여 함수를 호출하면 탄창에는 (웃쪽부터) 다음과 같은 내용이 들어있다.

- c 함수를 완료할 때 실행할 되돌아갈 명령주소
- 탄창에 보관한 사용자방식등록기주소
- 하드웨어오유코드

2) 례외처리기로의 진입과 복귀

이미 설명한것처럼 례외처리기를 구성하는 C함수명은 항상 처리기이름앞에 do_라는 앞붙이를 붙인것이다. 이 함수의 대부분은 하드웨어오유코드와 례외벡토르를 current의 프로쎄스서술자에 보관하고 이 프로쎄스에 적절한 신호를 전달한다. 이것은 다음과 같이 한다.

```
current->tss.error_code = error_code;
current->tss.trap_no= vector;
force_sig(sig_number, current) ;
```

현재프로쎄스는 례외처리기를 마친 직후에 신호를 처리한다. 프로쎄스가 자기의 신호취급기를 등록하였다면 사용자방식에 있는 처리기에서 신호를 처리하고 그렇지 않으면 핵심부방식에서 처리한다. 후자의 경우 핵심부는 보통 프로쎄스를 소멸한다. 례외처리기가 보내는 신호의 목록은 이미 표 5-1에서 보여주었다.

례외처리기는 항상 례외가 사용자방식에서 발생했는지 아니면 핵심부방식에서 발생했는지 검사한다. 후자의 경우라면 프로쎄스가 체계호출에 잘못된 파라메터를 전달했는지 검사한다. 핵심부방식에서 발생한 다른 례외는 핵심부오유이다. 이 경우 례외처리기는 핵심부가 오동작하고있다고 판단하고 하드디스크에 있는 자료를 망가뜨리지 않으려고 die()함수를 호출해서 모든 CPU등록기의 내용을 작업대로 출력하고(이것을 핵심부웁스(kernel oops)라고 한다.) do_exit()를 호출하여 current프로쎄스를 완료한다.

례외처리기를 구성하는 C함수가 끝나면 다음 기호언어코드부분으로 조종권이 넘어간다.

```
addl $8, %esp
jmp ret_from_exception
```

이 코드는 탄창에 보관한 사용자방식등록기의 주소와 하드웨어오유코드를 탄창에서 꺼낸 후 ret_from_exception()함수로 jmp명령을 실행한다.

11. 새치기처리

대부분의 례외는 례외를 발생시킨 프로쎄스에 Unix신호를 보내서 처리한다. 따라서 례외가 발생할 때 취하는 실질적인 행동은 프로쎄스가 신호를 받을 때까지 미루어져 핵심부는 례외를 빠르게 처리할수 있다. 그러나 이런 접근방법은 새치기에는 통하지 않는다. 왜냐면 새치기와 관련된 프로쎄스(례를 들면 자료전송을 요청한 프로쎄스)가 보류되고 한참후에 전혀 무관계한 프로쎄스가 실행중일 때 새치기를 수신하는 경우가 빈번하기때문이다. 따라서 현재프로쎄스에 Unix신호를 보내면 안된다.

새치기처리방법은 새치기의 류형에 따라 달라진다.여기서는 새치기를 크게 3가지 류형으로 구분한다.

- **입출력새치기**

여러 입출력장치는 주의(attention)를 필요로 한다. 해당 새치기처리기는 장치를

조사하여 어떤 동작을 취해야 하는지 결정해야 한다.

o 시계새치기

국부APIC시계이든 외부시계이든 일부시계는 새치기를 발생시킨다. 이 류형의 새치기는 핵심부에 지정한 시간간격이 지났음을 알려준다. 이 새치기는 대부분 입출력새치기로 처리하며 시계새치기 특유의 특징을 설명한다.

o 처리기간새치기

한 CPU가 다중처리기체계에 있는 다른 CPU에 새치기를 발생시킨다. 《처리기간새치기처리》에서 이 류형의 새치기를 다룬다.

1) 입출력새치기처리

일반적으로 입출력새치기처리기는 동시에 여러 장치에 봉사를 제공할수 있도록 유연해야 한다. 레를 들어 PCI모선구조에서는 여러 장치가 같은 IRQ선을 공유할수 있다. 이것은 새치기벡토르만으로는 어느 장치에서 새치기가 발생하였다고 확실히 말할수 없다는 의미이다. 표 5-3의 실례에서는 USB포구와 음성카드에 똑같은 벡토르 43번을 할당하고있다. 그렇지만 오래된 PC구조에 있는 몇몇 하드웨어장치는(실례로 ISA장치) 자기의 IRQ선을 다른 장치와 공유할 경우 제대로 동작하지 않는다. 새치기처리기는 다음 두가지 별개의 방식을 리용하여 유연하게 동작한다.

o IRQ공유

새치기처리기는 새치기봉사루틴(ISR, Interrupt Service Routine)을 여러개 실행한다. 각 ISR는 IRQ선을 공유하는 장치 하나와 련관된 함수이다. 어떤 창치가 IRQ를 발생시켰는지 미리 알수 없기때문에 어떤 장치가 주의를 필요로 하는지 확인하려고 각 ISR를 모두 실행한다. 그래서 새치기를 발생시킨 장치이면 필요한 모든 동작을 한다.

o IRQ동적할당

가능한 나중까지 장치가 IRQ선을 사용하는것을 미룬다. 레를 들어 사용자가 유연성장치로 접근할 때에만 유연성장치의 IRQ선을 할당한다. 이렇게 해서 같은 IRQ선을 공유할수 없는 하드웨어장치라도(동시가 아니라면) 여러 장치가 같은 IRQ벡토르를 사용할수도 있다.

새치기가 발생할 때 해야 하는 모든 작업이 똑같이 급하지는 않다. 사실 새치기처리기자체는 여러 종류의 일을 하기에 적합하지 않다. 새치기처리기가 실행중일 때에는 해당 IRQ선으로 오는 새치기신호를 무시하므로 시간이 오래 걸리면서도 중요하지 않은 작업은 나중으로 미루어야 한다. 가장 중요한 사실은 새치기처리기를 실행할 당시의 현재프로쎄스는 항상 TASK_RUNNING 상태에 있어야 하며 그렇지 않으면 체계가 멈출수도 있다는 점이다. 따라서 새치기처리기에서는 디스크입출력작업처럼 차단(blocking)을 일으킬수 있는 코드를 실행하면 안된다. Linux는 새치기가 발생할 때 처리할 작업을 3부류로 분류한다.

o 중요함

PIC에 새치기에 대한 응답을 보내거나 PIC나 장치조종기를 다시 프로그람화하거나

장치와 처리기가 동시에 접근하는 자료구조를 갱신하는 등의 작업. 이것은 가능한 빨리 수행해야 하기때문에 중요하며 빨리 처리할수 있다. 중요한 작업은 새치기처리기내에서 마스크할수 있는 새치기를 금지한 상태에서 즉시 실행한다.

o 중요하지 않음

처리기만 접근하는 자료구조를 갱신하는 등의 작업.(례를 들면 건반의 건을 눌렀을 때 건반주사코드를 읽는 작업) 이 작업 역시 빨리 처리할수 있으며 새치기처리기내에서 새치기를 허용한 상태에서 즉시 실행한다.

o 중요하지 않으며 미룰수 있음

완충기내용을 프로쎄스의 주소공간으로 복사하는 등의 작업.(례를 들면 건반완충기를 말단으로 리용하는 프로쎄스에 전달하는 작업) 이 작업은 핵심부동작에 영향을 미치지 않으면서 긴 시간동안 미룰수 있다. 이와 련관된 프로쎄스만이 계속 자료를 기다릴뿐이다. 중요하지 않으면서 미룰수 있는 작업은 《쏘프트 IRQ와 소과제, 하반부》에서 다루는 별도의 함수를 리용하여 수행한다.

어떤 회로에서 발생한 새치기인지 상관없이 모든 입출력새치기처리기는 다음 4가지 기본적인 작업을 수행한다.

1. IRQ값과 등록기내용을 핵심부방식탄창에 보관한다.
2. IRQ선을 봉사하는 PIC에 응답신호를 보내서 다음새치기를 발생시킬수 있게 한다.
3. IRQ를 공유하는 모든 장치와 관련된 새치기봉사루틴을 실행한다.
4. ret_from_intr()주소로 이행하여 완료한다.

```
#define BUILD_INTERRUPT(name, nr)  \
ENTRY(name)                                        \
    pushl $nr-256;                           \
    SAVE_ALL                                 \
    call smp_/**/name;                   \
    jmp ret_from_intr;
```

IRQ선의 상태와 새치기가 발생할 때 실행할 함수를 나타내려면 여러 서술자가 필요하다. 그림 5-3은 하드웨어회로와 새치기를 처리하는 쏘프트웨어함수의 관계를 도식적으로 보여준다. 이 함수는 다음에 설명한다.

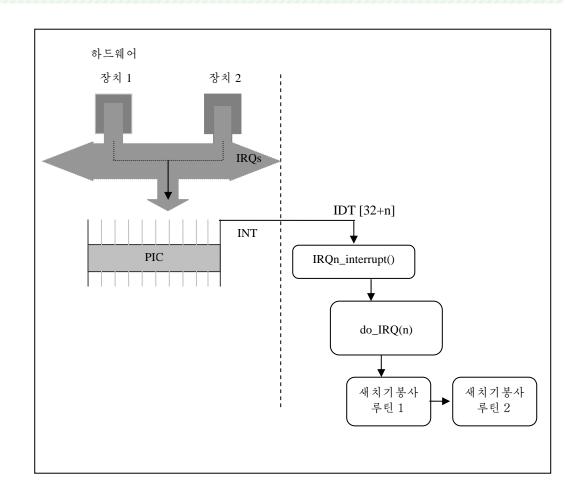



그림 5-3. 새치기운전

2) 새치기벡토르

표 5-2에서 보는것처럼 물리적인 IRQ에 32-238 범위에 있는 아무 벡토르나 할당할수 있다. Linux는 체계호출을 구성하는데 128번 벡토르를 사용한다.

IBM호환 PC구조에서 몇몇 장치는 정적으로 특정IRQ선에 련결해야 한다.

· 간격시계(interval timer)장치를 IRQ0선에 련결해야 한다.

· 종속8259A PIC를 IRQ2선에 련결해야 한다.(이제는 향상된 PIC를 사용하지만 Linux는 여전히 8259A방식 PIC를 지원한다).

· 외부수학보조처리기를 IRQ13선에 련결해야 한다.(최근에 나온 80x86처리기에서는 이런 장치를 더는 사용하지 않지만 Linux는 여전히80386모형을 지원한다.)

· 일반적으로 입출력장치를 그 수가 제한된 IRQ선으로 련결할수 있다.(사실은 IRQ공유를 지원하지 않는 이전 PC를 사용하면 이미 있는 하드웨어장치와 IRQ가 충돌해서 새 카드를 설치할수 없을수도 있다.)

표 5-2. Linux에서 새치기벡토르

벡토르범위	용 도
0-19(0x0-0x13)	마스크블가능한 새치기와 례외
20-31(0x14-0x1f)	예약
32-127(0x20-0x7f)	외부새치기(IRQ)
128(0x80)	체계호출을 위한 프로그람에 의한 례외
129-238(0x81-0xEE)	외부새치기(IRQ)
239(0xEF)	국부APIC시간새치기
240-250(0xF0-0xFA)	Linux가 마지막에 사용하기 위해 예약함
251-255(0xFB-0xFF)	프로쎄스사이새치기

IRQ를 설정할수 있는 장치의 IRQ선을 선택하는 방법은 다음 3가지가 있다.

・몇몇 하드웨어설정꼭지(jumper)설정을 통해 선택.(매우 오래된 장치카드에만 해당한다.)

・ 장치를 설치할 때 실행하는 장치에 따라 오는 유틸리티프로그람으로 선택. 이 프로그람은 사용자가 사용가능한 IRQ번호중에서 선택하도록 하거나 체계를 조사해서 스스로 사용가능한 값을 결정한다.

・ 체계를 시작할 때 실행하는 하드웨어적인 통신규약에 의해 선택. 이런 체계에서 주변장치는 어떤 새치기선을 사용할 준비가 되여있는지 선언한다. 최종값은 협상을 통해 가능한 충돌을 줄이는쪽으로 정한다. 일단 이 작업이 끝나면 각 새치기처리기는 장치의 특정입출력포구에 접근하는 함수를 사용하여 할당된 IRQ값을 읽을수 있다. 례를 들어 주변장치호상련결(PCI:Peripheтal Component Interconnect)표준과 호환하는 장치용구동프로그람의 경우 pci_read_config_byte() 같은 함수를 사용하여 장치의 설정공간(configuration space)에 접근할수 있다. 표 5-3은 어떤 pc에 존재할수 있는 장치와 IRQ의 임의의 배치를 보여준다.

핵심부는 새치기를 허용하기 전에 입출력장치에 대응하는 IRQ번호를 찾아야 한다. 그렇지 않다면 례를 들어 SCSI장치에 대응하는 벡토르가 몇번인지 모르는 상태에서 어떻게 장치에서 오는 새치기신호를 처리할수 있을것인가? 각 장치구동프로그람은 초기화를 할 때 이런 대응관계를 만들어야 한다.

표 5-3. 입출력장치에 IRQ를 할당하는 실례

IRQ	INT	하드웨어장치
0	32	시계
1	33	건반
2	34	PIC중첩
3	35	두번째 직렬포구
4	36	첫번째 직렬포구
6	38	유연성디스크
8	40	체계박자
10	42	망대면부
11	43	USB포구, 음성카드
12	44	PS/2마우스
13	45	수학보조프로쎄스
14	46	EIDE디스크조종기의 첫번째 사슬
15	47	EIDE디스크조종기의 두번째 사슬

3) IRQ자료구조

상태전이가 일어나는 복잡한 연산을 론의하기 전에 먼저 핵심자료를 어디에 보관하는지 리해하면 도움이 된다. 따라서 여기서는 새치기처리를 지원하는 자료구조와 이것을 여러 서술자에서 어떻게 배치하는지 설명한다. 그림 5-4는 IRQ선의 상태를 나타내는 주요서술자사이의 관계를 도식적으로 보여준다.(이 그림에서는 쏘프트IRQ와 소과제, 하반부를 처리하는데 필요한 자료구조는 표시하지 않았다. 이 자료구조는 이 뒤부분에서 다룬다.)

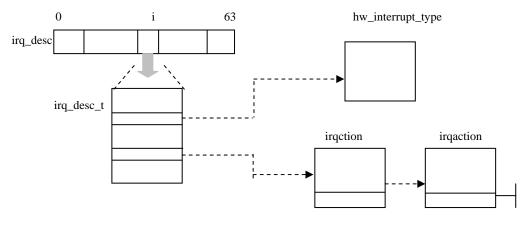



그림 5-4. IRQ 서술자

irq_desc는 NR_IRQS(보통은 224)개의 irq_desc_t서술자의 배렬이다. 이 서술자에는 다음과 같은 마당이 들어있다.

status

IRQ선의 상태를 나타내는 기발의 접합이다.(표 5-4 참고)

handler

IRQ선을 처리하는 PIC회로를 나타내는 hw_interrupt_type서술자에 대한 지적자이다.

action

IRQ가 발생할 때 호출할 새치기봉사루틴을 나타낸다. 이 마당은 IRQ와 관련된 irqaction서술자목록의 첫번째 요소를 가리킨다. irqaction서술자는 뒤에서 간단하게 설명한다.

표 5-4. IRQ선의 상태를 나타내는 기발

기발이름	설 명
IRQ_INPROGRESS	해당 IRQ용조종기를 실행중이다.
IRQ_DISABLED	장치구동프로그람에서 일부러 IRQ선을 금지하였다.
IRQ_PENDING	해당선에서 IRQ가 발생하여 PIC가 응답했지만 핵심부가 이것을 아직 처리하지 않았다.
IRQ_REPLAYIRQ	IRQ선을 금지했지만 PIC가 그 전에 발생한 IRQ에 대한 응답을 하지 않았다.
IRQ_AUTODETECT	핵심부가 하드웨어장치를 조사하는 과정에 IRQ선을 사용중이다.
IRQ_WAITING	핵심부가 하드웨어장치를 조사하는 과정에서 IRQ선을 사용중이다. 그리고 해당새치기가 아직 발생하지 않는다.
	80x86기본방식에서는 사용하지 않는다.
IRQ_LEVEL	사용하지 않는다.
IRQ_MASKED	80x86기본방식에서는 사용하지 않는다.
IRQ_PER_CPU	

depth

IRQ 선을 허용하고있으면 0, 한번이라도 금지했으면 정수값이다. disable_irq()나 disable_irq_nosync()함수를 호출할 때마다 이 마당을 증가시킨다. 이 함수는 값을 증가시키기 전에 depth마당의 값이 0이면 IRQ선을 금지시키고 IRQ_DISABLED기발을 설정한다. 반대로 enable_irq()함수를 호출할 때마다 이 마당을 감소시키며 depth가 0이 되면 IRQ선을 허용하고 IRQ_DISABLED기발을 지운다.

lock

IRQ서술자에 대한 접근을 직렬화할 목적으로 사용하는 스핀잠그기(spin lock)이다. 체계초기화과정에서 init_IRQ()함수는 각 IRQ주요서술자의 status마당을 IRQ_DISABLED로 설정한다. 다음 init_IRQ()함수는 IDT에 있는 림시새치기문을 새것으로 교체한다. 이것은 다음문장으로 한다.

```
for (i = 0; i < (NR_VECTORS - FIRST_EXTERNAL_VECTOR); i++) {
        int vector = FIRST_EXTERNAL_VECTOR + i;
        if (i >= NR_IRQS)
                break;
        if (vector != SYSCALL_VECTOR)
                set_intr_gate(vector, interrupt[i]);
```

이 코드는 interrupt 배렬에서 새치기문으로 설정할 새치기처리기의 주소를 알아낸다. IRQ용새치기처리기의 이름은 IRQn_interrupt()이다. (뒤에 나오는 《새치기처리기를 위한 등록기보관》 참고)

새치기문중 일부는 전혀 사용하지 않고 어떤것은 다중처리기체계에서만 사용하며 결국 항상 사용하는것은 일부이다. 따라서 새치기문가운데서 일부를 최종값으로 설정하지만 나머지는 그렇지 않는다. 좀더 자세히 살펴보자.

· 처음 16개의 IRQ(벡토르 32-47)용문을 최종값으로 설정한다.

· 다중처리기체계에서는 처리기간새치기용문과 국부APIC시계새치기용문도 옳바로 설정한다.(뒤에 나오는 《새치기봉사 루틴》 참고)

· 벡토르 128번은 체계호출의 프로그람작성에 의한 례외용으로 사용하기때문에 다치지 않는다.

· 나머지문은 pci모선에 련결된 장치에서 발생하는 새치기용으로 예약되여있다. 이 경우 irq_desc의 handler항목을 no_irq_type이라는 빈값처리기로 초기화한다.

Linux는 이 장의 시작부분에서 설명한 8259A소자외에도 SMP IO-APIC나 PILX4의 내부8259 PIC, SGI의 Visual Workstation Cobalt (IO-)APIC 같은 여러 PIC회로를 지원한다. Linux는 이런 장치를 똑같은 방식으로 다루려고 PIC의 이름과 PIC표준메쏘드 7개로 이루어진 PIC객체를 사용한다. 이런 객체지향적인 접근의 우점은 장치가 체계에 있는 PIC의 종류에 신경쓰지 않아도 된다는것이다. 각 장치에서 보기에는 새치기발생지가 옳바른 조종기로 련결되여있다. PIC객체를 정의하는 자료구조는 hw_interrupt_type이다.(hw_irq_controller라고도 부른다.)

구체적으로 설명하기 위해 8259A PIC 두개를 가진 단일처리기체계로서 표준IRQ 16개를 지원하는 콤퓨터가 있다고 가정하자. 이 경우 irq_desc_t 16개 각각에 있는 handler마당은 8259A PIC의 서술자인 i8259A_irq_type 변수를 가리킨다. 이 변수는

교육성 교육정보쎈터

다음과 같이 초기화된다.

```
struct hw_interrupt_type i8259A_irq_type = {
    "XT-PIC",
    startup_8259A_irq,
    shutdown_8259A_irq,
    enable_8259A_irq,
    disable_8259A_irq,
    mask_and_ack_8259A,
    end_8259A_irq,
    NULL
};
```

이 구조체의 첫번째 마당은 PIC의 이름인 《XT-PIC》이다. 그 다음에는 PIC를 프로그람화하는데 사용하는 함수 6개에 대한 지적자가 있다. 처음 두 함수는 각각 각 소자의 IRQ선을 시작하고 끝내는 함수이다. 그러나 8259A소자의 경우 이 두 함수는 새치기선을 허용하고 금지하는 세번째와 네번째 함수와 동일하다. mask_and_ack_8259A()함수는 8259A의 입출력포구에 적절한 값을 보내서 수신한 IRQ에 대한 응답을 한다. end_8259A_irq()함수는 IRQ선의 새치기처리기를 끌마칠때 호출하는 함수이다. 마지막 set_affinity 메쏘드는 NULL이다. 이 함수는 다중처리기체계에서 CPU와 특정IRQ와의 친화력(affinity) 즉 CPU가 특정IRQ를 처리할수 있도록 허가하는것을 정의할 때 사용한다.

앞서 설명한것처럼 여러 장치가 한 IRQ를 공유할수도 있다. 따라서 핵심부는 특정하드웨어장치와 특정새치기를 참조하는 irqaction서술자를 관리한다. 각 서술자에는 다음마당이 들어있다.

handler

입출력장치용새치기봉사루틴을 가리킨다. 이것이 여러 장치가 똑같은 IRQ를 공유할수 있게 하는 핵심마당이다.

flags

IRQ선과 입출력장치사이의 관계를 설명한다.(표 5-5 참고)

name

입출력장치의 이름이다.(/proc/interrupts 파일을 읽으면 나오는 봉사중인 IRQ 목록에서 이 이름을 볼수 있다.)

표 5-5. I rqaction서술자의 기발

기발 이름	설명
SA_INTERRUPT	새치기를 금지한 상태에서 처리기를 실행해야 한다.
SA_SHIRQ IRQ	선을 다른 장치와 공유할수 있도록 허용하는 장치이다.
SA_SAMPLE_RANDOM	이 장치에서는 사건이 불규칙적으로 일어난다고 생각해도 된다. 따라서 핵심부의 임의의(random) 수자발생기에서 이 장치를 사용할수 있다.(사용자는 /dev/random과 /dev/urandom장치파일에서 임의의 수자를 가져오는 방식으로 이 기능을 리용할수 있다.)

dev_id

입출력장치에서 사용하는 비공개(private)마당이다. 일반적으로 입출력장치자체를 구별하거나 (례를 들어 주번호(major number)와 부번호(minor number)값일수 있다.)장치구동프로그람의 자료를 가리킨다.

next

irqaction서술자목록의 다음항목을 가리킨다. 목록에 있는 항목은 같은 IRQ를 공유하는 하드웨어장치를 가리킨다.

마지막으로 체계에 있는 각 CPU마다 하나씩 NR_CPU개 입구점을 가지는 irq_stat배렬이 있다. 각 입구점의 형은 irq_cqustat_t이며 핵심부가 모든 CPU가 현재하는 일을 추적할수 있도록 몇개의 계수기와 기발을 가진다. 여기 있는 중요한 마당을 접근할 때에는 CPU의 론리번호(즉 배렬에서의 색인)를 파라메터로 받는 몇가지 마크로를 사용한다.

자세히 살펴보면 local_irq_count(n) 마크로는 배렬의 n번째 항목의 __local_irq_count 마당을 반환한다. 이 마당은 CPU에서 몇개의 새치기처리기가 쌓여있는지 즉 시작하였다가 아직 끝나지 않은 새치기처리기가 몇개인지 나타내는 계수기이다.

4) 다중처리기체계에서 IRQ분배

Linux는 대칭형다중처리(SMP, Symmetric Multiprocessing)모형을 준수한다. 이 말은 기본적으로 핵심부는 특정CPU를 다른 CPU보다 먼저 선택해서는 안된다는 의미이다. 그 결과 핵심부는 하드웨어장치에서 오는 IRQ신호를 모든 CPU사이에서 원형(round-robin)방식으로 분배하도록 노력한다. 따라서 모든 CPU는 입출력새치기를 봉사하는데 거의 비슷한 비률로 실행시간을 소모한다.

앞서 《향상된 프로그람가능한 새치기조종기》에서 다중 APIC체계는 CPU사이에서 IRQ신호를 동적으로 분배하는 복잡한 수법을 가지고있다고 하였다. 따라서 원형분배방책을 구성하기 위해 Linux핵심부가 해야 할 일은 거의 없다.

체계기동을 하는 동안 동작하는 CPU는 setup_IO_APIC_irqs() 함수를 실행하여

입출력APIC소자를 초기화한다.

여기서 입출력하드웨어장치에서 발생하는 모든 IRQ신호값을 가장 낮은 우선순위 방책에 따라 체계에 있는 각 CPU로 전달하도록 새치기재지정표에 있는 입구점 24개를 설정한다. 다음 체계기동을 하는 동안 모든 CPU는 setup_1ocal_APIC()함수를 실행하여 국부APIC를 초기화한다. 특히 각 소자의 과제우선순위등록기(TPR, Task priority Register)를 CPU가 우선순위에 상관없이 어떤 종류의 IRQ신호라도 처리하겠다는것을 의미하는 고정된 값으로 설정한다. Linux핵심부는 초기화를 한 후 이 값을 절대 바꾸지 않는다.

모든 과제우선순위등록기는 값이 같기때문에 모든 CPU는 우선순위가 항상 같다, 다중APIC체계는 이 동등상태를 파괴하기 위해 앞에서 설명한 국부APIC의 중재우선순위등록기의 값을 사용한다. 이 값은 새치기가 발생한 후에 자동으로 바뀌기때문에 IRQ신호를 모든 CPU사이에서 균등하게 분배한다.

간단히 말해서 하드웨어장치가 IRQ신호를 발생시키면 다중 APIC체계는 CPU중 하나를 선택해서 해당 국부 APIC로 신호를 전달하여 해당 CPU에 새치기를 발생시킨다. 다른 모든 CPU는 이 사건을 전달받지 않는다. 이 모든 일이 하드웨어에 의해서 이루어지므로 핵심부는 다중 APIC체계를 초기화한 후에는 이에 신경쓸 필요가 없다.

5) 새치기처리기를 위한 등록기보관

CPU는 새치기를 수신하면 IDT의 해당 문에서 찾은 주소에 있는 코드를 실행하기 시작한다.(앞에서 나온 《새치기와 례외의 하드웨어적인 처리》 참고)

다른 문맥절환과 마찬가지로 등록기를 보관하기 위해 핵심부개발자는 좀 복잡한 코딩작업을 해야 한다. 등록기보관과 복구는 기호언어코드를 리용해서 해야 하는데 이런 연산을 할 때 처리기는 C함수를 호출하고 C함수에서 돌아오는것처럼 여기기때문이다. 여기에서는 등록기를 다루는 기호언어작업을 먼저 설명하고 이어서 호출하는 C함수를 위해 필요한 사항을 살펴보자. 등록기보관은 새치기처리기에서 처음으로 수행하는 과제이다. 이미 설명한것처럼 IRQn에 대한 새치기처리기는 IRQn_interrupt이고 이 주소는 해당 IDT입구점에 있는 새치기문에 들어있다.

단일처리기체계에서는 각 IRQ번호마다 하나씩 똑같은 BUILD_IRQ마크로를 16번 사용하여 서로 다른 새치기처리기 16개의 입구점(entry point)을 만든다. 다중처리기체계에서는 이 마크로를 14×16번 사용하여 총합 새치기처리기 224개의 입구점을 만든 각 마크로는 다음기호언어코드로 확장된다.

```
IRQn_interrupt:
pushl $n-256
```

jmp common_interrupt는 새치기에 해당하는 IRQ번호에서 256을 던 값을 탄창에 보관한다.이렇게 하면 모든 새치기처리기에서 이 번호를 참조하는 똑같은 코드를 사용할 수 있다. BUILD_COMMON_IRQ 마크로에 공통코드가 있으며 이 마크로는 다음기호

언어코드로 확장된다.

```
common_interrupt:
SAVE_ALL
call do_IRQ
jmp $ret_from_intr
```

SAVE_ALL 마크로는 다음코드가 된다.

```
cld
push %es
```

IRQ 번호에서 256을 빼면 짝수가 된다. 홀수는 체계호출을 나태내는것으로 예약하고있다.

```
push %ds
pushl %eax
pushl %ebp
pushl %edi
pushl %esi
pushl %edx
pushl %ecx
pushl %ebx
movl $__ KERNEL_DS,%edx
movl %edx, %ds
movl %edx, %es
```

SAVE_ALL은 조종장치가 이미 자동으로 보관한 eflags와 cs, eip, ss, esp등록기(앞에서 살펴본 《새치기와 례외의 하드웨어적인 처리》 참고)를 제외하고 새치기처리기에서 사용할수 있는 모든 CPU등록기를 탄창에 보관한다. 그리고 나서 ds와 es를 핵심부자료토막의 값으로 설정한다. BUILD_COMMON_IRQ는 등록기를 보관한 후 do_IRQ()함수를 호출한다. do_IRQ()함수에서 ret명령을 실행하면(즉 함수를 끝마치면) ret_from_intr()로 조종이 넘어간다.(뒤에 나오는 《새치기와 례외로부터 복귀》 참고)

6) do_IRQ()함수

do_IRQ()는 새치기와 련결된 모든 새치기봉사루틴을 실행하기 위해 호출하는 함수이다. 이 함수를 시작할 때 핵심부탄창에는 디음과 같은 내용이 우에서부터 차례로 들어있다.

・do_IRQ()를 마치고 돌아갈 주소(ret_from_intr()의 시작주소)
・SAVE_ALL로 보관한 등록기값들
・변환한 IRQ번호

・새치기가 발생했을 때 조종장치가 자동으로 보관한 등록기

C콤파일러는 모든 파라메터를 탄창의 맨우에 배치하므로 do_IRQ()함수는 다음과 같이 정의한다.

```
unsigned int do_IRQ (struct pt_regs regs)
```

pt_regs구조체에는 마당이 15개 있다.

・처음 9개 마당은 SAVE_ALL로 보관한 등록기에 해당한다.

・10번째 마당은 orig_eax로 참조하는 마당으로 변환한 IRQ번호이다.

・나머지마당은 조종장치가 자동으로 보관한 등록기에 해당한다.

do_IRQ()함수는 다음코드와 동일하다.

```
int irq=regs.orig_eax & 0xff;
spin_lock(&(irq_desc[irq].lock));
irq_desc[irq].bandler->ack(irq);
irq_desc[irq].status &=~(IRQ_REPLAY | IRQ_QAITING);
irq_desc[irq].status | =IRQ_PENDING;
if (!(irq_desc[irq].status & (IRQ_DISABLED | IRQ_INPROGRESS))&&
irq_desc[irq].action){
irq_desc[irq].status | =IRQ_INPROGRESS  ;
do {
irq_desc[irq].status & = ~IRQ_PENDING;
spin_unlock(&(irq_desc[irq].lock));
handle_IRQ_event(irq,&regs, irq_desc[irq].action);
spin_lock(&(irq_desc[irq].lock));
} while (irq_desc[irq].status & IRQ_PENDING);
irq_desc[irq].status &=~IRQ_INPROGRESS;
}
irq_desc[irq].handler->end(irq);
spin_unlock(&(irq_desc[irq].lock));
if (softirq_pending(smp_processeor_id()))
do_softirq();
```

맨 먼저 do_IRQ()함수는 탄창을 통해 전달한 파라메터에서 IRQ벡토르를 알아내서 이것을 국부변수 irq에 보관한다. 이 값을 irq_desc배렬(IRQ주서술자)에 있는 옳바른 항목을 접근하는 색인으로 사용한다.

pc_regs구조체에는 함수를 마치고 돌아갈 주소인 ret_from_intr()은 들어가지 않는다. 왜냐면 C콤파일러는 돌아갈 주소가 탄창의 맨 꼭대기에 있다고 생각하면 파라메터를 지정하는 명령을 만들기때문이다.

핵심부는 IRQ주서술자에 접근하기 전에 해당하는 스핀잠그기(spin lock)를 얻는다. 스핀잠그기는 서로 다른 CPU에서 동시에 접근하는것을 막는다.(단일처리기체계에서 spin_lock()함수는 아무일도 하지 않는다.) 다중처리기체계에서는 같은 종류의 다른 새치기가 발생할수 있고 이때 다른 CPU가 새로 발생한 새치기를 처리할수 있으므로 스핀잠그기가 꼭 필요하다. 스핀잠그기가 없다면 여러 CPU가 동시에 주IRQ서술자를 접근할수 있다. 나중에 살펴보지만 이런 상황은 전적으로 피해야 한다.

스핀잠그기를 얻은 후 이 함수는 주IRQ서술자에 있는 ack메쏘드를 호출한다. 단일처리기체계에서 이에 해당하는 함수는 mask_and_ack_8259A()함수로써 PIC에 새치기에 대한 응답을 하고 그 IRQ선을 금지한다. IRQ선을 금지함으로써 처리기를 완료할때까지 CPU가 이 종류의 새치기를 더는 받아들이지 않게 한다. do_IRQ()함수는 국부새치기를 금지한 상태에서 실행한다는것을 기억해야 한다. 사실 새치기처리기는 IDT의 새치기문을 통해 호출하므로 CPU조종장치는 자동으로 eflags등록기의 IF기발을 0으로 지운다. 그렇지만 이 새치기용새치기봉사루틴을 실행하기 전에 핵심부가 국부새치기를 다시 허용할수도 있다는 사실을 곧 보게 될것이다.

다중처리기체계에서는 일이 훨씬 복잡하다. 새치기의 류형에 따라 새치기에 응답(acknowledge)하는것을 ack메쏘드에서 할수도 있고 새치기처리기를 완료할 때까지 미룰수도 있다.(즉 end메쏘드에서 응답을 하는것이다.) 어느 경우라도 국부APIC는 처리기를 완료할 때까지 이 종류의 새치기를 더는 받아들이지 않는다고 생각해도 된다. 다음에 발생하는 같은 종류의 새치기를 다른 CPU가 받아들일수는 있다.(주IRQ서술자에 있는 스핀잠그기가 이런 상황을 해결해준다.)

다음으로 do_IRQ()함수는 주IRQ서술자에 있는 몇가지 기발을 초기화한다. 새치기에 응답을 했지만 실제 처리하지 않았으므로 IRQ_PENDING기발을 설정한다. 또한 IRQ_WAITING과IRQ_REPLAY기발을 지운다.(아직은 이 기발에 신경쓰지 않아도 된다.)

이제 do_IRQ()는 정말로 새치기를 처리해야 하는지 검사한다. 새치기가 발생해도 아무일도 하지 않아야 하는 경우는 다음과 같은 3가지이다.

IRQ_DISABLED가 설정된 경우

해당 IRQ선을 금지한 상태이더라도 CPU가 do_IRQ()함수를 호출할수도 있다. 오유가 있는 주기판때문에 PIC에서 IRQ선을 금지한 경우라도 가짜새치기가 발생할수도 있다.

IRQ_INPROGRESS가 설정된 경우

다중처리기체계에서 다른 CPU가 이전에 발생한 같은 종류의 새치기를 처리하고있는중일수 있다. 그렇다면 이번 새치기처리를 해당 CPU로 미루는것은 어떻겠는가? 이것이 바로 Linux가 하는 일이다. 이것은 장치구동프로그람의 새치기봉사루틴을 재진입가능하게 작성하지 않아도 되므로(직렬로 실행하기때문에)핵심부구조를 좀 더 간단하게 만

들어준다. 다른 CPU에 넘겨서 일이 없어진 CPU는 하드웨어캐쉬를 더는 더럽히지 않고 빨리 이전에 하던 일로 돌아갈수 있다. 이것은 체계성능에 도움이 된다. CPU가 새치기와 련관된 새치기봉사루틴을 실행할 때마다 IRQ_INPROGESS기발을 설정하고 do_IRQ()함수는 실제작업을 시작하기 전에 이 기발을 검사한다.

irq_desc[irq].action이 NULL인 경우

이 경우는 새치기와 련결된 새치기봉사루틴이 없을 때이다. 보통 이 경우는 핵심부가 하드웨어장치를 조사할 때에만 발생한다.

이 3가지 경우에 해당하지 않는다면 새치기를 처리해야 한다. do_IRQ()함수는 IRQ_INPROGRESS기발을 설정한 후 순환구간을 돌기 시작한다. 반복을 할 때마다 이 함수는 IRQ_PENDING기발을 지우고 새치기스핀잠그기를 해제하고 handle_IRQ_event()(《새치기봉사루틴》에서 설명한다.)를 호출하여 새치기봉사루틴을 실행한다. handle_IRQ_event()가 끝나면 do_IRQ()는 스핀잠그기를 다시 얻고 IRQ_PENDING기발을 검사한다. 이 기발이 지워졌다면 더는 해당 새치기가 발생하여 다른 CPU로 전달되지 않은것이므로 순환을 완료한다. 반대로 IRQ_PENDING기발이 설정되였다면 이 CPU에서 handle_IRQ_event()를 실행하는중에 다른 CPU에서 이 새치기용으로 do_IRQ()를 실행한것이다. 따라서 do_IRQ()는 순환을 다시 반복하고 새로 발생한 새치기를 처리한다.

이제 do_IRQ()함수는 새치기봉사루틴을 이미 실행했거나 할 일이 없어서 완료하려고 한다. 이 함수는 주IRQ 서술자에 있는 end메쏘드를 호출한다. 단일처리기체계에서 해당 함수는 end_8259A_irq()로 IRQ선을 다시 허용한다.(가짜로 새치기가 발생한 경우가 아니라면) 다중처리기체계에서 end메쏘드는 ack메쏘드에서 새치기에 응답하지 않은 경우 응답을 한다.

IRQ_PENDING은 계수기가 아닌 기발이기때문에 두번째로 발생한 새치기만을 알수 있다. do_IRQ()의 순환고리를 반복하는 동안에 이 이상 발생한 새치기는 잃어버린다.

마지막으로 do_IRQ()는 스핀잠그기를 해제한다. 이제 어려운 일은 끝났다. 돌아가기 전에 이 함수는 실행하기를 기다리고있는 미룰수 있는 핵심부함수가 있는지 검사한다.(나중에 나오는 《 쏘프트IRQ와 소과제, 하반부 》 참고) 작업이 대기중이면 do_softirq()함수를 호출한다. do_IRQ()함수가 끝나면 ret_from_intr()함수로 조종이 넘어간다.

7) 잃어버린 새치기 부활

do_IRQ()함수는 작고 간단하지만 대부분의 경우 제대로 동작한다. 실제로 IRQ_PENDING과 IRQ_INPROGRESS, IRQ_DISABLED기발은 하드웨어가 오동작하더라도 새치기를 정확하게 처리하도록 한다. 그런데 다중처리기체계에서는 아주 부드럽게만 동작하는것은 아니다.

한개 CPU가 어떤 IRQ선을 허용하고있다고 가정하자. 하드웨어장치에서 해당

IRQ선에 새치기요청을 발생시키고 다중 APIC체계에서 새치기를 처리할 CPU로 이 CPU를 선택한다고 하자. 이때 이 CPU가 새치기에 대해 응답을 보내기 전에 다른 CPU가 해당 새치기를 마스크해서 막아버리면 결과적으로 IRQ_DISABLED기발이 설정된다. 바로 다음에 앞의 CPU에서 대기중인 새치기를 처리하기 시작하고 do_IRQ()함수는 새치기에 대해 응답을 한 후 IRQ_DISABLED기발이 설정된것을 발견하여 새치기봉사루틴을 실행하지 않은채 완료한다. 그 결과 IRQ선을 금지하기 전에 발생한 새치기는 잃어버리게 된다.

이런 경우에 대처하기 위해 enable_irq()함수는 IRQ선을 다시 허용할 때 잃어버린 새치기가 있다면 강제로 하드웨어가 해당 새치기를 발생시키게 한다.

이 함수는 잃어버린 새치기가 있다는 사실을 IRQ_PENDING기발을 검사하여 알아낸다. 새치기처리기를 빠져나갈 때 이 기발을 항상 지우므로 IRQ선이 금지된 상태에서 이 기발이 설정되여있다면 새치기가 발생하여 이에 응답을 했지만 이것을 처리하지 않았음을 의미한다. 이 경우 새로 새치기를 발생시켜야 한다. 이것은 국부APIC가 자기에 새치기를 발생시키도록 하여 처리한다.(뒤에 나오는 《처리기사이 새치기처리》참고) IRQ_REPLAY기발은 이런 자기에게 가는 새치기가 한번만 발생하도록 하는 역할을 한다. 앞에서 do_IRQ()함수가 새치기를 처리하기 시작할 때 이 기발을 지운다는 점을 기억할것이다.

8) **새치기봉사루틴**

앞에서 본것처럼 새치기봉사루틴은 장치에 특수한 작업을 구성한다. 새치기처리기가 ISR을 실행할 때에는 handle_IRQ_event()함수를 호출한다. 이 함수는 근본적으로 다음목록에 나오는 단계대로 실행한다.

a. irq_enter()함수를 호출하여 이것을 실행하는 CPU의 irq_stat입구점에 있는 __local_irq_count마당을 증가시킨다.(CPU에 새치기처리기가 몇개 쌓여있는지 알려면 이전에 나온 《IRQ자료구조》를 참고하시오.) 이 함수는 대역으로 금지하지 않은 새치기도 검사한다.

b. SA_INTERRUPT기발을 설정하지 않고있으면 sti기호언어명령을 사용하여 국부새치기를 허용한다.

c. 다음코드를 통해 새치기와 련관된 각 새치기봉사루틴을 실행한다.

```
do {
action->handler(irq,action->dev_id, regs);
action=action->next;
}while(action);
```

순환을 시작할 때 action은 새치기를 수신할 때 실행할 작업인 irqaction자료구조의 목록의 시작을 가리킨다.(앞에 나온 [그림 5-4] 참고)

d. cli기호언어명령을 사용하여 국부새치기를 금지한다.

e. irq_exit()를 호출하여 이것을 실행하는 CPU의 irq_stat입구점에 있는 __local_irq_count마당을 감소시킨다.

모든 새치기봉사루틴은 똑같은 파라메터로 작업을 한다.

Irq

IRQ번호

dev_id

장치식별자

regs

새치기가 발생한 직후에 보관한 등록기를 포함하는 핵심부방식탄창령역에 대한 지적자

첫번째 파라메터는 한 ISR이 여러 IRQ선을 다룰수 있도록 하고 두번째 파라메터는 한 ISR이 같은 IRQ를 사용하는 여러 장치를 조심스럽게 구별하게 한다. 마지막파라메터는 ISR가 새치기된 핵심부조종경로의 실행문맥에 접근할수 있게 한다. 실제로 대부분의 ISR는 이 파라메터를 사용하지 않는다.

주IRQ서술자의 SA_INTERRUPT기발로 do_IRQ()함수가 ISR을 호출할 때 새치기를 금지해야 하는지 허용해야 하는지 지정한다. 새치기를 허용하거나 금지한 상태에서 호출하는 ISR는 그 내부에서 새치기상태를 반대로 바꿀수도 있다. 단일처리기체계에서는 cli(clear interrupt, 새치기금지)나 sti(set interrupt, 새치기허용)라는 기호언어명령을 사용하여 새치기상태를 바꿀수 있다. 다중처리기체계에서 모든 CPU의 새치기를 허용하거나 금지하는것은 훨씬 더 복잡하다. ISR의 구조는 ISR가 다루는 장치의 특성에 따라 달라진다.

9) IRQ선의 동적할당

앞에서 《새치기벡토르》에서 설명한것처럼 벡토르 몇개만 특정장치용으로 예약하고 나머지는 동적으로 처리한다. 따라서 하드웨어장치가 IRQ공유를 지원하지 않는다고 하더라도 여러 장치가 같은 IRQ선을 사용할수 있다. 이 기법은 하드웨어장치를 직렬로 동작하게 하여 동시에 하나만 IRQ를 소유하게 하는것이다.

IRQ선을 사용하려는 장치를 활성화하기 전에 해당 구동프로그람은 request_irq()를 호출한다. 이 함수는 새로 irqaction서술자를 만들어 이것을 넘겨준 파라메터값으로 초기화한다. 그런 다음 setup_irq()함수를 호출하여 이 서술자를 옳바른 IRQ목록에 삽입한다. setup_irq()함수가 오유값을 돌려주면 이것은 그 IRQ선을 이미 사용하고있는 다른 장치가 새치기공유를 지원하지 않는다는 의미이며 장치구동프로그람은 동작을 멈추어야 한다. 장치가 동작을 마치면 구동프로그람은 free_irq()함수를 호출하여 IRQ목록에서 서술자를 제거하고 기억기령역을 해제한다.

간단한 실례를 들어 이것이 어떻게 동작하는지 보자. 프로그람이 체계에 있는 첫번째 유연성디스크에 해당하는 장치파일인 /dev/fd0에 접근한다고 하자. 프로그람은 이것을 /dev/fd0파일에 직접 접근하거나 여기에 있는 파일체계를 탑재해서 할수 있다.

유연성디스크조종기에는 보통 IRQ6을 할당하며 이에 따라 유연성디스크구동프로그람은 다음과 같은 요청을 한다.

request_irq(6,floppy_interrupt,

SA_INTERRUPT(SA_SAMPLE_RANDOM, “floppy”, NULL);

이것을 보면floppy_interrupt()새치기봉사루틴은 반드시 새치기를 금지한 상태에서 실행한다는 점과(SA_INTERRUPT기발을 지정) IRQ공유를 지원하지 않는다는점을(SA_SHIRQ기발을 지정하지 않음)알수 있다. SA_SAMPLE_RANDOM기발을 지정한것은 유연성디스크에 대한 접근은 임의로 발생하는 사건의 좋은 원천으로 핵심부의 임의의 수자생성기에서 이것을 사용한다는것을 나타낸다. 유연성디스크의 동작을 마치면(/dev/fd0으로 요청한 입출력동작이 끝나거나 파일체계를 탑재해제하거나) 구동프로그람은 IRQ6을 해제한다.

free_irq(6, NULL);

핵심부는 irqaction서술자를 옳바른 목록에 삽입할 때에는 IRQ번호인 irq_nt와 이전에 할당한 irqaction서술자의 주소인 new를 파라메터로 하여 setup_irq()함수를 호출한다. 이 함수는 다음과 같이 동작한다.

a. 다른 장치가 이미 irq_nr IRQ를 사용중인지 검사한다. 사용중이면 두 장치가 모두 irqaction서술자에서 SA_SHIRQ기발을 지정하여 IRQ선을 공유할수 있는지 여부를 검사한다. IRQ선을 사용할수 없으면 오유값을 반환한다.

b. *new(new가 가리키는 새 irqaction서술자)를 irq_desc[irq_nr]->action이 가리키는 목록의 맨끝에 추가한다.

c. 다른 장치가 같은 IRQ를 공유하지 않으면 *new flags마당에서 IRQ_DISABLED와 IRQ_AUTODETECT, IRQ_INPROGRESS기발을 지우고 irq_desc[irq_nr]->handler가 가리키는 PIC객체에 있는 startup메쏘드를 호출하여 IRQ신호를 허가한다.

여기에 체계초기화코드에서 가져온 setup_irq()를 사용하는 실례가 있다. 핵심부는 time_init()함수에서 다음 명령을 실행하여 간격시계장치의 irq0서술자를 초기화한다.

struct irqaction irq0=

{timer_interrupt, SA_INTERRUPT, 0, “timer”, NULL,};

setup_irq(0, &irq0);

먼저 irqaction형의 irq0변수를 초기화한다. Handler마당을 timer_interrupt()함수의 주소로, flags마당을 SA_INTERRUPT로, name마당을 《timer》로, 마지막마당을 NULL로 지정하여 dev_id값을 사용하지 않는다고 표시한다. 다음으로 핵심부는 setup_irq()를 호출하여 irq0을 IRQ0에 련결된 irqaction서술자의 목록에 삽입한다.

10) 처리기사이새치기처리

다중처리기체계에서 Linux는 다음에 나오는 다섯가지 종류의 처리기사이새치기를

정의한다.([표 5-2]참고)

CALL_FUNCTION_VECTOR(벡토르 0xfb)

자기를 제외한 모든 CPU에 보내서 이것들 CPU가 파라메터로 전달하는 함수를 실행하게 만든다. 해당하는 새치기처리기의 이름은 call_function_interrupt()이다. 파라메터로 전달하는 함수는 례를 들어 다른 CPU를 멈추게 할수도 있고 기억기형범위등록기(MTRR: Memory Type Range Register)의 내용을 설정하게 만들수도 있다. 보통 smp_call_function()함수를 사용하여 이 함수를 호출하는 자기를 제외한 모든 CPU에 이 새치기를 보낸다.

RESCHEDULE_VECTOR(벡토르 0xfc)

CPU가 이 종류의 새치기를 수신하면 해당 처리기인 reschedule_interrupt()는 새치기에 응답하는 일만 한다. 새치기에서 돌아갈 때 자동으로 재순서짜기가 일어난다.(뒤에 나오는 《새치기와 례외로부터 복귀》참고)

INVALIDATE_TLB_VECTOR(벡토르 0xfd)

자기를 제외한 모든 CPU에 보내서 이 CPU들이 각각의 변환참조완충기(translation lookaside buffer)를 무효화하도록 한다. 해당 처리기의 이름은 invalidate_interrupt()로 TLB입구점을 비운다.

ERROR_APIC_VECTOR(벡토르0xfe)

이 새치기는 절대로 발생하면 안된다.

SPURIOUS_APIC_VECTOR(벡토르0xff)

이 새치기는 절대로 발생하면 안된다.

다음에 나오는 함수를 사용하여 처리기사이새치기(IPI:interprocessor interrupt)를 손쉽게 발생시킬수 있다.

send_IPI_all()

호출하는 자기를 포함한 모든 CPU에 IPI를 전송한다.

send_IPI_allbutself()

호출하는 자기를 제외한 모든 CPU에 IPI를 전송한다.

send_IPI_self()

호출하는 자기에 IPI를 전송한다.

send_IPI_mask()

비트마스크(bit mask)로 지정한 일련의 CPU에 IPI를 전송한다.

BUILD_SMP_INTERRUPT마크로를 사용하여 처리기사이새치기처리기의 기호언어코드를 만든다. 이 코드는 BUILD_IRQ마크로가 만드는 코드와 거의 같다.(앞에서 본 《새치기처리기를 위한 등록기보관》참고)

각 처리기사이새치기마다 서로 다른 고수준처리기가 있다. 이 처리기의 이름은 저수준처리기의 이름앞에 smp_라는 앞붙이를 붙인것이다. 례를 들어 RESCHEDULE_

VECTOR처리기사이새치기용 저수준처리기인 reschedule_interrupt()가 호출하는 고수준처리기의 이름은 smp_reschedule_interrupt()이다. 각 고수준처리기는 국부APIC로 처리기사이새치기에 대한 응답(acknowledge)을 하고 발생한 새치기에 따르는 특수한 작업을 한다.

11. 쏘프트IRQ와 소과제, 하반부

앞에서 《새치기처리》에서 핵심부가 수행하는 작업중 몇가지는 급하지 않다고 하였다. 이 작업은 필요하다면 한참후로 미룰수도 있다. 새치기처리기는 새치기봉사루틴을 직렬로 실행하고 종종 새치기처리기가 완료하기 전에는 해당 새치기가 더는 발생하지 않아야 한다. 반대로 미룰수 있는 작업은 모든 새치기를 허용한 상태에서 실행할수 있다. 이것을 새치기처리기에서 분리하면 핵심부가 새치기에 반응하는 시간을 줄이는데 도움이 된다. 이것은 새치기요청을 불과 수ms내에 처리할것이라고 기대하는 시간에 민감한 많은 응용프로그람에 매우 중요한 특성이다.

Linux에서는 이런 요청에 대해 미룰수 있고 새치기가능한 핵심부함수(줄여서 미룰수 있는 함수(deferrable function)) 3종류를 제공한다. 이것은 쏘프트 IRQ(soft half)와 소과제(tasklet), 하반부(bottom half)이다. 이 세 종류의 미룰수 있는 함수는 비록 다른 방식으로 동작하지만 서로 밀접하게 련관되여있다. 소과제를 쏘프트IRQ우에서 구성하고 하반부는 소과제를 사용하여 구성한다. 사실 핵심부코드내에서는 종종 모든 종류의 미룰수 있는 함수를 쏘프트IRQ라고 지정한다.

일반적으로 같은 CPU에서 한 쏘프트IRQ가 다른 쏘프트IRQ를 멈출수 없다. 쏘프트IRQ를 바탕으로 구성하는 소과제와 하반부에도 같은 규칙이 적용된다. 그렇지만 다중처리기체계에서는 서로 다른 CPU에서 미룰수 있는 함수 여러개를 동시에 실행할수 있다. 동시성의 정도는 표 5-6에서 볼수 있는것처럼 미룰수 있는 함수의 류형에 따라 다르다.

표 5-6. 쏘프트 IRQ와 소과제, 하반부의 차이

지연함수	동적할당	동시성
쏘프트IRQ	아니	같은 종류의 쏘프트IRQ를 여러 CPU에서 동시에 실행할수 있다.
소과제	예	다른 종류의 소과제를 여러 CPU에서 동시에 실행할수 있다.
하반부	아니	하반부는 여러 CPU에서 동시에 실행할수 없다.

쏘프트IRQ와 하반부는 정적으로만 할당하지만(즉 콤파일시에 정의한다.) 소과제는 실행시(례를들어 핵심부모듈을 적재할 때)에 할당하고 초기화할수도 있다.

많은 쏘프트IRQ는 같은 종류더라도 항상 여러 CPU에서 동시에 실행할수 있다. 일

교육성 교육정보쎈터

반적으로 쏘프트IRQ는 재실행가능한 함수로 자기의 자료구조를 스핀잠그기를 사용하여 명백하게 보호해야 한다.

소과제는 자기에 대해서는 항상 직렬로 실행해야 한다는 점에서 쏘프트IRQ와 다르다. 즉 한 종류의 소과제를 동시에 두 CPU에서 실행할수 없다. 그렇지만 서로 다른 종류의 소과제를 여러 CPU에서 동시에 실행할수 있다. 소과제를 직렬화하면 소과제를 처리하는 함수를 재실행가능하게 만들지 않아도 되므로 장치구동프로그람개발을 단순하게 할수 있다.

마지막으로 하반부는 대역으로 직렬화한다. 어떤 CPU에서 하반부를 실행하고있으면 다른 CPU에서는 비록 다른 종류라도 어떠한 하반부도 실행할수 없다. 이것은 매우 강력한 제한사항으로 다중처리기체계에서 Linux핵심부의 성능을 떨어뜨린다. 사실 Linux는 단지 호환성을 위해 계속 하반부를 지원하고있을뿐이며 장치구동프로그람개발자들이 하반부를 사용하는 오래된 구동프로그람를 소과제를 사용하도록 교체하기를 바라고있다. 따라서 먼 후날에는 Linux에서 하반부가 사라질것이다. 어느 경우이든 미룰수 있는 함수를 직렬로 실행해야 한다. 같은 CPU에서는 어떤 미룰수 있는 함수라도 다른 미룰수 있는 함수와 중첩하여 실행할수 없다.

일반적으로 미룰수 있는 함수와 관련하여 네 종류의 동작을 수행한다.

초기화

새로 미룰수 있는 함수를 정의한다. 이 작업은 보통 핵심부자체를 초기화할 때 한다.

활성화

미룰수 있는 함수를 대기중(pending)으로 즉 다음에 미룰수 있는 함수들을 실행할 때 이것을 수행하라고 표시한다. 활성화는 언제나 할수 있다.(새치기를 처리하는중이라도 가능하다.)

마스크

미룰수 있는 함수를 선택적으로 금지하여 이것을 활성화한 경우라도 핵심부가 이것을 실행하지 않게 한다.

실행

대기중인 미룰수 있는 함수를 같은 종류의 다른 모든 미룰수 있는 함수와 함께 실행한다.

《쏘프트IRQ》에서 설명하지만 특별히 지정한 시점에서만 이것을 실행한다.

활성화와 실행은 어느 정도 서로 묶여있다. 특정 CPU에서 활성화한 미룰수 있는 함수는 반드시 같은 CPU에서 실행해야 한다. 이 규칙이 체계성능에 도움이 된다는것을 나타내는 어떤 자명한 근거는 없다. 리론적으로는 미룰수 있는 함수를 활성화한 함수와 묶으면 CPU 하드웨어캐쉬를 더 잘 활용하게 된다. 결국 활성화한 핵심부스레드에서 접근한 자료구조를 미룰수 있는 함수에서도 사용할것이라고 생각할수 있다. 그렇지만 미룰수 있는 함수를 실행하기까지는 오랜 시간이 걸릴수도 있기때문에 해당 함수를 실행할

당시에는 캐쉬에 그와 련관된 캐쉬행이 남아있지 않을수도 있다. 나가서 함수를 한 CPU와 묶으면 해당 CPU만 매우 바쁘고 다른 CPU는 한가한 상태가 발생할수 있어 잠재적으로 위험한 방법일수 있다.

1) 쏘프트 IRQ

Linux에서는 제한된 수의 쏘프트IRQ(softirq)를 사용한다. 대부분의 경우 소과제으로도 충분하고 소과제는 재진행가능하지 않아도 되므로 훨씬 작성하기 쉽다.

사실 표 5-7에서 서술한것처럼 현재 네 종류의 쏘프트 IRQ만 정의한다.

표 5-7. Linux2.6에서 사용하는 쏘프트 IRQ

쏘프트IRQ	색인(우선순위)	설 명
HI_SOFTIRQ	0	우선순위가 높은 소과제와 하반부를 처리한다.
NET_TX_SOFTIRQ	1	망카드를 통해서 패키지를 전송한다.
NET_RX_SOFTIRQ	2	망카드에서 패키지를 수신한다.
TASKLET_SOFTIRQ	3	소과제를 처리한다.

Linux에서 사용하는 쏘프트의 색인은 우선순위를 나타낸다. 핵심부는 쏘프트IRQ함수를 색인 0부터 실행하기때문에 색인값이 낮다는것은 우선순위가 높다는것을 의미한다.

쏘프트IRQ를 나타내는 핵심자료구조는 softirq_vec 배렬로서 softirq_action형요소 32개를 포함한다. 쏘프트IRQ의 우선순위는 배렬내에 있는 해당 softirq_action에 해당하는 색인이다. 표 5-7에서 보는것처럼 처음 4개 입구점만 실제로 사용한다. softirq_action 자료구조는 마당 두개로 이루어진다. 하나는 쏘프트IRQ함수이고 다른 하나는 쏘프트IRQ함수에서 필요로 할수 있는 일반적인 자료구조에 대한 지적자이다.

《IRQ자료구조》에서 이미 설명한 irq_stat배렬에는 핵심부가 쏘프트IRQ(쏘프트IRQ를 기반으로 하는 소과제와 하반부도 포함하여)를 구성하는데 필요한 여러가지 마당이 있다. 배렬의 각 항목은 CPU 하나에 대응한다. 여기에 들어있는 마당은 다음과 같다.

__softirq_pending 마당은 softirq_action 구조체를 가리킨다(대기중인 쏘프트IRQ). softirq_pending 마크로를 사용하여 이 마당에 쉽게 접근할수 있다.

__local_bh_count 마당은 쏘프트IRQ의 실행을 금지한다.(소과제와 하반부도 함께) local_bh_count 마크로를 사용하여 이 마당에 쉽게 접근할수 있다. 이 값이 0이면 쏘프트IRQ를 허용하고 이 마당이 홀수이면 쏘프트IRQ를 금지한다. 1ocal_bh_disable 마크로는 이 값을 증가시키고 1ocal_bh_enable마크로는 감소시킨다. 핵심부가 1ocal_bh_disable을 2번 호출하면 1ocal_bh_enable 역시 2번 호출해야 쏘프트IRQ를

다시 허용할수 있다.

__ksoftirq_task 마당은 미룰수 있는 함수의 실행을 담당하는 ksoftirqd_CPUn 핵심부스레드의 프로쎄스서술자를 보관한다.(이 스레드는 CPU마다 하나씩 있으며 ksoftirqd_CPUn에서 n은 CPU 색인을 나타낸다. 이것은 《쏘프트IRQ핵심부스레드》에서 설명한다.) ksoftirqd_task 마크로를 사용하여 이 마당을 접근할수 있다.

open_softirq()함수는 쏘프트IRQ의 초기화를 맡는다. 이 함수는 쏘프트IRQ색인과 실행할 쏘프트IRQ함수에 대한 지적자, 쏘프트IRQ함수에서 필요로 할수 있는 자료구조에 대한 지적자 이렇게 세개의 파라메터를 받는다. open_soflirq()는 softirq_vec 배렬에서 해당하는 입구점만 초기화한다. __CPU_raise_softirq마크로를 호출하여 쏘프트IRQ를 활성화한다. 이 마크로는 CPU번호인 CPU와 쏘프트 IRQ의 색인인 nr를 파라메터로 받고 softirq_pending(CPU)의 nr번째 비트를 1로 설정한다. CPU_raise_sofiirq() 함수는 __CPU_raise_softirq 마크로와 비슷하지만 ksoftirqd_CPUn핵심부스레드도 함께 깨울수도 있다는 점이 다르다.

핵심부코드의 일부지점에서만 대기중인 쏘프트IRQ가 있는지 검사한다. 현재는 다음경우에만 이것을 수행한다.(쏘프트IRQ를 검사하는 회수와 위치는 핵심부판본과 지원하는 하드웨어구조에 따라 달라질수 있다는데 주의하시오.)

- 1ocal_bh_enable 마크로로 쏘프트IRQ를 다시 허용할 때
- do_IRQ() 함수가 입출력새치기처리를 마칠 때
- smp_APIC_timer_interrupt()함수가 국부시계새치기처리를 마칠 때(《다중프로쎄스체계에서 시간관리구조》참고)
- 특별한 ksoftirqd_CPUn 핵심부스레드중 하나가 깨여날 때
- 망이음부카드를 통해 패키지를 수신할 때

각 검사지점마다 핵심부는 softirq_pending(CPU)을 읽는다. 이 마당이 빈값(NULL)이 아니면 핵심부do_softirq()를 호출하여 쏘프트 IRQ함수를 실행한다. 이것은 다음과 같이 동작한다.

1.이 함수를 실행하는 CPU의 론리번호 CPU를 알아낸다.

2.1ocal_irq_count(CPU)가 0이 아니면 그냥 돌아간다. 이 경우는 중첩된 새치기처리기를 완료할 때 do_softirq()를 호출한것이다. 이미 알고있는것처럼 미룰수 있는 함수는 새치기봉사루틴바깥에서 실험해야 한다.

3.Local_bh_count(CPU)가 0이 아니면 그냥 돌아간다. 이 경우는 미룰수 있는 함수를 금지한것이다.

4.IF 기발의 상태를 보관하고 이 기발을 지워서 국부새치기를 금지한다.

5. irq_stat의 softirq_pending(CPU)마당을 검사한다. 대기중인 쏘프트IRQ가 없다면 앞단계에서 보관한 IF기발의 값을 복구하고 돌아간다.

6.1ocal_bh_disable(CPU)을 호출하여 irq_stat의 local_bh_count(CPU)마당을

증가시킨다. 이렇게 하면 이후에 do_softirq()를 호출하면 쏘프트IRQ함수를 실행하지 않고 돌아가기때문에(앞의 3단계 참고) 해당 CPU에서 미룰수 있는 함수를 효과적으로 직렬로 실행하게 된다.

7. 다음순환을 실행한다.

```
pending = softirq_pending(CPU);
softirq_pending(CPU) = 0;
mask = ~0;
do {
mask &= ~pending;
asm("sti" );
for (i=0; pending; pending >>=1, i++)
if (pending & 1)
softirq_vec[i].action(softirq_vec+i);
asm("cli");
pending = softirq_pending(CPU);
} while (pending & mask);
```

코드에서 보는것처럼 이 함수는 국부변수 pending에 대기중인 쏘프트IRQ를 보관하고 softirq_pending(CPU)마당을 0으로 설정한다. 순환을 반복할 때마다 이 함수는 다음 일을 한다.

a. 국부변수 mask를 갱신한다. 이 변수는 이번에 실행하는 do_softirq()함수에서 이미 실행한 쏘프트IRQ의 색인을 보관한다.

b. 국부새치기를 허용한다.

c. 대기중인 모든 쏘프트IRQ를 실행한다.(안쪽순환)

d. 국부새치기를 금지한다.

e. 국부변수 Pending에 softirq_pending(CPU) 마당의 내용을 다시 보관한다. 쏘프트IRQ함수를 실행하는 동안 새치기처리기에서 또는 쏘프트IRQ함수에서 CPU_raise_softirq()를 호출했을수 있다.

ㅂ. 이번에 실행하는 do_softirq()에서 처리하지 않은 쏘프트IRQ가 새로 활성화되였다면 순환을 다시 돈다.

local_bh_count(CPU)마당을 감소시켜 쏘프트IRQ를 다시 허용한다. 국부변수 pending을 검사한다. 이 값이 0이 아니라면 이번에 실행한 do_softirq()에서 처리한 쏘프트IRQ가 다시 활성화된것이다. 다시 do_softirq()함수를 실행할수 있도록 ksoftirqd_CPUn핵심부스레드를 깨운다.

8. 4단계에서 보관한 IF기발의 상태를 복구하고(국부새치기를 금지하거나 허용한다.) 함수를 마친다.

2) 쏘프트 IRQ핵심부스레드

최근판본의 핵심부에는 각 CPU마다 자기만의 ksoftirqd_CPUn핵심부스레드(n은 CPU의 론리번호이다.)가 있다. 각 ksoftirqd_CPUn핵심부스레드는 ksoftirqd()함수를 실행한다. 이 함수는 다음순환를 실행한다.

```
for(;;) {
set_current_state(TASK_INTERRUPTIBLE);
schedule();
/* 이제 TASK_RUNNING 상태이다. */
while (softirq_pending(CPU)) {
do_softirq();
if (current->need_resched)
schedule();
}
}
```

이 핵심부스레드가 깨여나면 softirq_pending(n)마당을 검사하여 필요하다면 do_softirq()를 호출한다.ksoftirqd_CPUn핵심부스레드는 민감한 일종의 거래(tradeoff)문제에 대한 해결책이다.

쏘프트IRQ함수는 자기를 다시 활성화할수 있다. 실제로 망환경을 담당하는 쏘프트IRQ와 소과제용쏘프트IRQ에서 이런 일을 한다. 망카드에서 파케트가 마구 들어오는 등의 외부사건이 발생하면 매우 짧은 주기마다 쏘프트IRQ를 활성화할수 있다.

쏘프트IRQ가 계속해서 대량으로 발생하는 경우 생길수 있는 문제의 가능성을 핵심부스레드를 도입하여 해결한다. 핵심부스레드가 없다면 개발자는 다음 두가지 방책을 대안으로 생각할수 있다.

첫번째 방책은 do_softirq()를 실행하는 동안에 발생한 새로운 쏘프트IRQ를 무시하는것이다. 다른 말로 하면 do_softirq()함수는 함수를 시작할 때 어떤 쏘프트IRQ가 대기중인지 확인한 후 그 함수들을 실행한다. 그리고 나서 대기중인 쏘프트IRQ를 다시 검사하지 않고 완료한다. 그러나 이 해결책으로는 충분하지 않다. 쏘프트IRQ함수가 do_softirq()를 실행하는 도중에 쏘프트IRQ가 다시 활성화되였다고 하자. 최악의 경우 콤퓨터가 한가한 상태라도 다음시계새치기가 발생할 때까지 해당 쏘프트IRQ를 실행하지 않을것이다. 그 결과 망개발자들이 받아들이기 힘든 쏘프트IRQ의 지연이 발생할것이다.

두번째 방책은 대기중인 쏘프트IRQ를 계속해서 다시 검사하는것이다. do_softirq()함수는 계속해서 대기중인 쏘프트IRQ를 검사하고 대기중인것이 없을 때에만 완료한다. 이 방책으로는 망개발자를 만족할수는 있겠지만 일반적인 체계사용자에게는 불편하게 할것이다. 망카드에서 빠른 속도로 파케트가 밀려들어와 쏘프트IRQ함수가

계속 자기를 활성화하면 do_softirq()함수는 절대로 끝나지 않고 사용자방식프로그람은 멈추어 보일수 있다.

ksoftirq_CPUn핵심부스레드는 이런 어려운 문제를 해결하려 한다. do_softirq() 함수는 어떤 쏘프트IRQ가 대기중인지 확인하고 해당 함수를 실행한다. 만약에 이미 실행한 쏘프트IRQ가 다시 활성화되면 핵심부스레드를 활성화하고 완료한다. 이 핵심부스레드는 우선 순위가 낮기때문에 사용자프로그람을 실행할수 있으면서 콤퓨터가 한가한 상태이면 대기중인 쏘프트 IRQ를 빠르게 처리하게 된다.

3) 소과제

소과제(taskler)는 입출력구동프로그람에서 미룰수 있는 함수를 구성할 때 먼저 선택하는 방법이다. 이미 설명한것처럼 소과제는 HI_SOFTIRQ와 TASKLET_SOFTIRQ라는 두 쏘프트IRQ를 기반으로 구성한다. 같은 쏘프트IRQ에 각각 자기만의 함수를 가지는 여러 소과제를 련결할수 있다.

do_softirq()가 HI_SOFTIRQ에 있는 소과제를 TASKLET_SOFTIRQ에 들어있는 소과제보다 먼저 실행한다는 점을 제외하면 실제로 두 쏘프트IRQ사이에 아무런 차이가 없다.

소과제와 우선순위가 높은 소과제를 각각 taskler_vec과 taskler_hi_vec배렬에 보관한다. 둘다 taskler_head형요소를 NR_CPUS개 가진다. 각 요소는 소과제서술자(taskler descryiptor)의 목록를 가리키는 지적자로 이루어진다. 소과제서술자는 taskler_struct형의 자료구조로 여기에는 표 5-8에 보여준 마당이 들어간다.

표 5-8. 소과제서술자의 마당

마당명	설 명
Next	목록에 있는 다음서술자에 대한 지적자
State	소과제의 상태
Count	잠그기(lock)계수기
Func	소과제함수에 대한 지적자
Data	소과제함수가 사용할수 있는 unsigned long int형의 자료

소과제서술자의 state마당은 두 기발을 포함한다.

TASKLET_STATE_SCHED

이 기발을 설정하여 소과제가 대기중임을 나타낸다.(실행하기 위해 순서짜기를 함) 또한 소과제서술자를 tasklet_vec나 tasklet_hi_vec배렬에 있는 목록중 하나에 삽입했는지 여부도 나타낸다.

TASKLET_STATE_RUN

이 기발을 설정하여 소과제를 실행중임을 나타낸다. 단일처리기체계에서는 특정소과제를 실행중인지 아닌지 검사할 필요가 없기때문에 이 기발을 사용하지 않는다.

장치구동프로그람를 작성하면서 소과제를 사용하려고 한다고 하자. 그러면 무엇을 해야 할것인가?

무엇보다도 먼저 새로 tasklet_struct자료구조를 할당한 후 tasklet_init()함수를 호출해서 이것을 초기화해야 한다. 이 함수는 소과제서술자의 주소와 소과제함수의 주소, 선택적으로 옹근수파라메터를 받는다.

tasklet_disable_nosync()나 tasklet_disable()을 호출하여 선택적으로 소과제를 금지할수 있다. 두 함수 모두 소과제서술자의 count마당을 증가시키지만 후자의 함수는 이미 해당 소과제함수를 실행중이면 이것을 완료하기 전에는 돌아오지 않는다는 차이가 있다. 소과제를 다시 허용하려면 tasklet_enable()을 사용한다.

소과제를 활성화하려면 해당 소과제에 필요한 우선순위에 따라 tasklet_schedule()이나 tasklet_hi_schedule()함수를 호출한다. 두 함수는 매우 비슷하며 각 함수는 다음 작업을 수행한다.

1. TASKLET_STATE_SCHED기발을 검사한다. 이 기발이 설정되여있으면 완료한다.(그 소과제를 이미 순서짜기하였다.)

2. 그 함수를 실행하는 CPU의 론리번호를 알아낸다.

3. IF기발의 상태를 지정하고 이 기발을 지워서 국부새치기를 금지한다.

4. 소과제서술자를 tasklet_vec[CPU]나 tasklet_hi_vec[CPU]가 가리키는 목록의 맨앞에 추가한다.

5. CPU_raise_softirq()를 호출하여 TASKLET_SOFTIRQ나 HI_SOFTIRQ 쏘프트IRQ를 활성화한다.

6. 3단계에서 보관한 IF기발의 상태를 복구한다.(국부 새치기를 허용하거나 금지한다.)

마지막으로 소과제를 어떻게 실행하는지 보자. 앞절에서 일단 활성화하면 do_softirq()함수에서 쏘프트IRQ함수를 실행한다고 하였다. HI_SOFTIRQ쏘프트IRQ에 대한 쏘프트IRQ함수는 tasklet_hi_action()이고 TASKlET_SOFTIRQ에 대한 함수는 tasklet_action()이다. 이 두 함수 역시 매우 비슷하다. 각 함수는 다음 작업을 한다.

1. 그 함수를 실행하는 CPU의 론리번호를 알아낸다.

2. 국부새치기를 금지한다.

3. tasklet_vec[CPU]나 tasklet_hi_vec[CPU]에서 가리키는 목록의 주소를 국부변수목록에 보관한다.

4. tasklet_vec[CPU]나 tasklet_hi_vec[CPU]를 NULL주소로 설정한다. 따라서 순서짜기를 한 소과제서술자의 목록는 비워진다.

5. 국부새치기를 허용한다.

6. 목록이 가리키는 각 소과제서술자에 다음과 같은 일을 한다.

a. 다중처리기체계라면 소과제에 있는 TASKLET_STATE_RUN기발을 검사한다. 이 기발이 설정되여있으면 같은 종류의 소과제를 다른 CPU에서 실행하고있는것이므로 해당 소과제서술자를 tasklet_vec[CPU]나 tasklet_hi_vec[CPU]가 가리키는 목록에 다시 삽입하고 TASKLET_IRQ나 HI_SOFTIRQ 쏘프트IRQ를 다시 활성화한다. 이런 식으로 CPU에서 같은 종류의 소과제를 실행하지 않을 때까지 실행을 나중으로 미룬다.

b. TASKLET_STATE_RUN기발이 설정되여있지 않으면 다른 CPU에서 그 소과제를 실행하지 않는것이다. 다중처리기체계에서는 이 기발을 설정하여 소과제함수를 다른 CPU에서 실행할수 없게 된다.

c. 소과제서술자의 count마당을 보고 소과제를 금지하고있는지 여부를 검사한다. 금지하고있으면 그 소과제서술자를 tasklet_vec[CPU]나 tasklet_hi_vec[CPU]가 가리키는 목록에 다시 삽입하고 TASKLET_IRQ나 HI_SOFTIRQ 쏘프트IRQ를 다시 활성화한다.

d. 소과제를 허용하고있으면 TASKLET_STATE_SCHED기발을 지우고 소과제함수를 실행한다.

소과제함수가 자기를 다시 활성화하지 않는한 한번 활성화된 소과제는 소과제함수를 한번만 실행한다.

4) 하반부

하반부(bottom half)는 근본적으로 다른 CPU, 다른 종류의 하반부라도 다른 하반부와 동시에 실행할수 없는 우선순위가 높은 소과제이다. 최대한 하반부만 실행할수 있도록 global_bh_lock스핀잠그기를 사용한다.

Linux는 모든 하반부를 함께 모아서 관리하려고 bh_base표이라는 배렬을 리용한다. 이것은 하반부에 대한 지적자의 배렬로 각 류형의 하반부마다 하나씩 32개까지 입구점을 포함할수 있다. 실제로 Linux는 이중 절반가량을 사용하며 표 5-9에 그 목록을 주었다. 표에서 볼수 있는것처럼 하반부중 일부는 체계에 필수적으로 들어가지 않는 하드웨어장치나 IBM PC호환체계를 제외한 다른 가동환경에 특수한 하드웨어장치와 관련된것이다. 그러나 TIMER_BH와 TQUEUE_BH, SERIAL_BH, IMMEDIATE_BH는 여전히 광범하게 사용한다. TQUEUE_BH와 IMMEDIATE_BH하반부는 조금 후에 설명하고 TIMER_BH하반부는 8장에서 설명한다.

표 5-9. Linux 하반부

하반부	주변장치
TIMER_BH	시계
TQUEUE_BH	정기적인 과제대기렬
DIGI_BH	DigiBoard PC/Xe
SERIAL_BH	직렬포구
RISCOM8_BH	RISCom/8
SPECIALIX_BH	Specialix IO8+
AURORA_BH	Aurora다중포구카드(SPARC)
ESP_BH	Hayes ESP직렬카드
SCSI_BH	SCSI이음부
IMMEDIATE_BH	즉시실행과제대기렬
CYCLADES_BH	Cyclades Cyclom-Y직렬다중포구
CM206_BH	CD_ROM Phillips/LMS cm206디스크
MACSERIAL_BH	Power Mac의 직렬포구
ISICOM_BH	MultiTech의 ISI카드

bh_task_vec는 각 하반부마다 하나씩 32개의 소과제서술자를 포함하는 배렬이다. 핵심부를 초기화하는 동안 다음과 같이 이 소과제서술자를 초기화한다.

```
for (i=0; i<32, ++i)
tasklet_init(bh_task_vec+i, bh_action,i);
```

보통 하반부를 처음 호출하기 전에 먼저 이것을 초기화해야 한다. 초기화할 때에는 init_bh(n, routine)함수를 호출한다. 이 함수는 routine주소를 bh_base의 n번째 입구점에 넣는다. 반대로 remove_bh(n)는 bh_base 표에서 n번째 하반부를 제거한다.

mark_bh()함수를 사용하여 하반부를 활성화한다. 하반부는 우선순위가 높은 소과제이므로 make_bh(n)는 tasklet_hi_schedule(bh_task_vec+n)을 실행한다.

bh_action()은 모든 하반부에서 공통으로 사용하는 소과제함수이다. 이 함수는 하반부의 색인를 파라메터로 받아 다음단계를 실행한다.

1. 소과제를 실행하는 CPU의 론리번호를 알아낸다.

2. global_bh_lock 스핀잠그기를 누가 점유하고있는지 검사한다. 그렇다면 다른 CPU가 하반부를 실행하고있는중이므로 이 함수는 mark_bh()를 호출하여 하반부를 다시 활성화하고 완료한다.

3. 그렇지 않으면 globaLbh_1ock 스핀잠그기를 획득하여 체계에서 다른 하반부를 실행할수 없게 한다.

4. local_irq_count 마당이 0인지(하반부는 새치기봉사루틴바깥에서 실행해야 한다.)대역새치기를 허용하고있는지 검사한다. 이중 하나라도 해당하지 않으면 global_bh_Iock스핀잠그기를 해제하고 완료한다.

5. bh_base 배렬에 있는 해당 입구점에 있는 하반부함수를 호출한다.

6. global_bh_lock 스핀잠그기를 해제하고 완료한다.

5) 하반부확장

미룰수 있는 함수를 도입한 계기는 새치기처리와 관련한 몇개 안되는 함수가 작업을 미루는 방식으로 일을 할수 있게 하기 위해서이다. 이러한 접근방법은 두 방향으로 확장되여왔다.

・새치기를 처리하는 함수뿐아니라 일반핵심부함수도 하반부로 실행할수 있게 한다.

・한개의 하반부에 하나가 아닌 여러 핵심부함수를 련관시킬수 있게 한다.

과제대기렬(task queue)이라는것으로 함수그룹을 나타낸다. 과제대기렬은 표 5-10에 나타낸 마당을 포함하는 tq_struct구조체를 요소로 하는 목록이다.

표 5-10. tq_struct구조체의 마당

마당명	설　명
List	2중련결목록에 대한 련결
Sync	여러번 활성화하는것을 막기 위해 사용
Routine	호출할 함수
Data	함수에 전달할 파라메터

뒤에서 다시 보지만 입출력장치구동프로그람은 특정새치기가 발생하면 련관된 함수 여러개를 실행하기 위해서 과제대기렬을 사용한다.

새로 과제대기렬을 할당할 때에는 DECLARE_TASK_QUEUE마크로를, 과제대기렬에 새로운 함수를 추가할 때에는 queue_task()를 사용한다. run_task_queue()함수는 지정한 과제대기렬에 들어있는 모든 함수를 실행한다.

여기서는 특별한 과제대기렬 세개를 설명한다.

・tq_immediate과제대기렬은 IMMEDIATE_BH하반부가 실행하는것으로 표준하반부와 함께 실행할 핵심부함수를 포함한다. 핵심부는 tq_immediate과제대기렬에 함수를 추가할 때마다 mark_bh()를 호출하여 IMMEDIATE_BH하반부를 활성화한다. 이 과제대기렬은 do_softirq()를 호출하자마자 실행된다.

・tq_timer과제대기렬은 시계새치기가 발생할 때마다 활성화되는 TQUEUE_BH하반부가 실행한다. 뒤에서 살펴보지만 거의 10ms마다 한번씩 이것을 실행한다.

・tq_context과제대기렬은 하반부와 관련없고 keventd핵심부스레드가 실행한다.

schedule_task()는 함수를 과제대기렬에 추가하고 순서짜기가 다음에 실행할 프로쎄스로 keventd를 선택할 때까지 실행을 미룬다.

미룰수 있는 함수를 기반으로 하는 다른 과제대기렬에 비해 tq_context를 사용하는 것의 중요한 우점은 자유롭게 차단을 일으킬수 있는 작업을 할수 있다는 사실이다. 반면에 쏘프트IRQ는 (당연히 소과제와 하반부도) 핵심부개발자가 그 미룰수 있는 함수를 실행할 시점의 현재프로쎄스에 대해 어떤 가정을 할수도 없다는 점에서 새치기처리기와 비슷하다. 실제적인 관점에서 보면 쏘프트IRQ는 파일을 접근하거나 신호기를 획득하거나 대기대기렬에서 기다리는 등의 차단을 일으킬수 있는 작업을 할수 없다. 이에 대한 대가는 tq_context에서 실행하라고 순서짜기하면 꽤 오랜 시간이 지난 후에 그 함수를 실행한다는것이다.

12. 새치기와 례외에서 복귀

이제 새치기와 례외처리기의 완료단계를 살펴보자. 이 단계의 주요목적은 명확하게 어떤 프로그람의 실행을 재개하는것이지만 그전에 몇가지 고려해야 할 사항이 있다.

동시에 실행중인 핵심부조종경로의 수

하나밖에 없다면 CPU를 사용자방식로 돌려보내야 한다.

대기중인 프로쎄스절환요청

이런 요청이 있다면 핵심부는 프로쎄스를 순서짜기해야 하고 그렇지 않으면 현재프로쎄스에 조종권을 넘겨준다.

대기중인 신호

현재프로쎄스에 신호를 보냈다면 이것을 처리해야 한다.

핵심부에서 이 모든것을 구성하는 기호언어코드는 기술적으로 말하면 함수가 아니다. 왜냐면 이것을 호출한 함수로 조종권이 돌아가지 않기때문이다. ret_from_intr과 ret_from_exception, ret_from_sys_call, ret_from_fork라는 서로 다른 4개 입구점에 있는 코드가 이런 일을 한다. 쉽게 설명하기 위해 이것을 서로 다른 4개 함수라고 설명하겠다. 앞으로 종종 다음 4개 입구점을 함수라고 설명할것이다.

ret_from_exception()

0x80을 제외한 모든 례외처리기를 완료한다.

ret_from_intr()

새치기처리기를 완료한다.

ret_from_sys_call()

체계호출, 즉 0x80프로그람작성에 의한 례외에서 만든 핵심부조종경로를 완료한다.

ret_from_fork()

fork()와 vfork(), clone()체계호출을 완료한다.(자식프로쎄스에서만 사용한다.)

그림 5-5는 4개 입구점(entry point)의 일반적인 흐름을 나타낸다. 그림에서

ret_from_exception()과 ret_from_int:r()입구점은 갈아보이지만 실제는 그렇지 않다. 전자의 경우 핵심부는 례외를 발생시킨 프로쎄스의 서술자를 알고있지만 후자의 경우 새치기와 련관된 프로쎄스서술자가 없다. 그림에서는 입구점에 해당하는 표외에도 코드흐름과 기호언어코드를 더 쉽게 련관지을수 있도록 표 몇개를 추가하였다. 이제 매 경우 어떻게 완료하는지 자세히 살펴보자.

1) ret_from_exception() 함수

ret_from_exception()함수는 다음기호언어코드와 비슷하다.

```
ret_from_exception:
movl Ox30(%esp), %eax
movb 0x2C(%esp), %al
testl $ (0x000200003), %eax
jne ret_from_sys_call
restore_all:
popl %ebx
popl %ecx
popl %edx
popl %esi
popl %edi
popl %ebp
popl %eax
popl %ds
popl %es
addl $4, %esp
iret
```

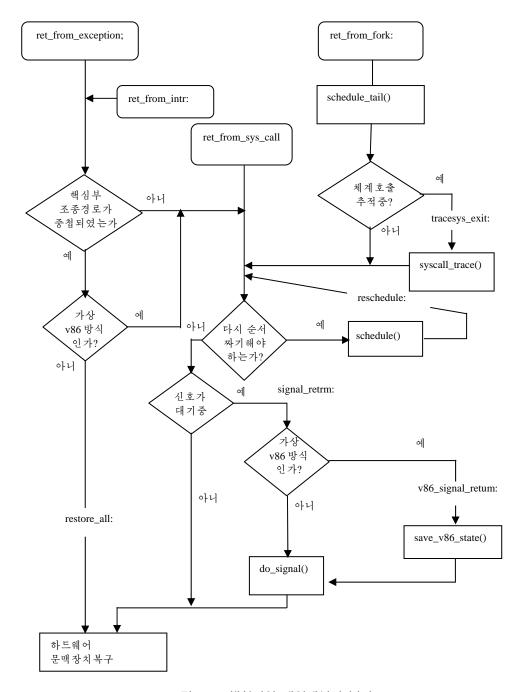



그림 5-5. 새치기와 례외에서의 복귀

이 함수는 례외가 발생할 때 탄창에 보관하는 cs와 eflags등록기의 값을 리용하여 중단된 프로그람이 사용자방식에서 동작하고있었는지 여부와 eflags의 VM기발이 설정되였는지 여부를 알아낸다. 둘중 하나라도 해당하면 ret_from_sys_call()함수로 이행한다. 그렇지 않으면 중단된 핵심부조종경로를 다시 시작해야 한다. 이 함수는 례외를 시작할 때 SAVE_ALL마크로로 보관했던 등록기를 복구하고 iret명령을 실행하여 중단된 프로그람으로 조종를 넘긴다.

이 기발는 프로그람을 가상8086방식(virtual-8086 mode, VM86 mode)에서 실행할수 있게 한다.

2) **ret_from_intr()함수**

ret_from_intr() 함수는 근본적으로 ret_from_exception()과 동등하다.

```
ret_from_intr:
movl $OxffffeOOO, %ebx
andl %esp, %ebx
jmp ret_from_exception
```

ret_from_exception()을 호출하기 전에 ret_from_intr()은 ebx등록기를 current 프로쎄스서술자의 주소로 설정한다. ret_from_exception()이 호출하기도 하는 ret_from_sys_call()함수는 ebx가 그 주소를 가지고있다고 가정하기때문에 이 과정은 꼭 필요하다. 한편 례외의 경우 ret_from_exception()을 실행하기 전에 례외처리기가 ebx를 current의 주소로 설정한다.(앞서 살펴본 《례외처리기를 위한 등록기보관》 참고)

3) **ret_from_sys_call()함수**

ret_from_sys_call() 함수는 다음기호언어코드로 되여있다.

```
ret_from_sys_call:
cli
cmpl $0, 20(%ebx)
jne reschedule
cmpl $0, 8 (%ebx)
jne signal_return
jmp restore_all
```

앞서 설명한것처럼 ebx등록기는 current 프로쎄스서술자를 가리킨다. 처음 나오는 cmpl명령어는 서술자내에서 편위 20에 있는 need_resched마당을 검사한다. 따라서 need_resched마당이 1이면 프로쎄스절환을 수행하는 schedule()함수를 호출한다.

```
reschedule:
```

```
call schedule
jmp ret_from_sys_all
```

프로세스서술자내에서 sigpending 마당의 편위는 8이다. 이 값이 빈값(null)이면 current는 탄창에 보관한 프로세스의 하드웨어문맥를 복구하여 사용자방식에서 실행을 재개한다. 빈값이 아니면 signal_return으로 뛰여넘기하여 current에서 대기중인 신호값을 처리한다.

```
signal_return;
sti
test1 $ (0x00020000), 0x30(%esp)
mov1 %esp, %eax
jne v86_signal_return
xor1 %edx, %edx
call do_signal
jmp restore_all
v86_signal_return;
call save_v86_state
mov1 %eax, %esp
xor1 %dex, %dex
call do_signal
jmp restore_all
```

새치기된 프로세스가 VM86방식에 있었다면 save_v86_state()함수를 호출한다. 다음으로 do_softirq()함수를 호출하여 대기중인 신호를 처리한다. 마침내 current는 사용자방식에서 실행을 할수 있다.

4) ret_from_fork()함수

fork()나 vfork(), clone()체계호출을 사용하여 생성한 자식프로세스는 생성직후 ret_from_fork()함수를 실행한다. 이 코드는 다음기호언어코드와 비슷하다.

```
ret_from_fork;
push1 %ebx
call schedule_tail
add1 $4, %esp
mov1 $0xffffe000, %ebx
and1 %esp, %ebx
testb $0x02, 24(%ebx)
jne tracesys_exit
jmp ret_from_sys_call
```

```
tracesys_exit;
call syscall_trace
jmp ret_from_sys_call
```

시작할 때 ebx등록기는 자식으로 교체되는 프로쎄스의 서술자(일반적으로 부모의 프로쎄스서술자)주소를 보관하고있다. 이 값을 schedule_tail()함수의 파라메터로 전달한다. 이 함수를 마친 후 ebx를 current의 프로쎄스서술자주소로 다시 설정한다. 다음으로 ret_from_fork()함수는 current의 ptrace마당(프로쎄스서술자의 편위 24에 위치한다.)의 값을 검사한다. 이 마당이 빈값이 아니면fork()나 vfork(), clone()체계호출을 추적하고있는중이므로 syscall_trace()함수를 호출하여 오유수정을 하는 프로쎄스에 알려준다.

제 2 절. 입출력장치관리

가상파일체계는 각 장치에 적합한 방식으로 읽기, 쓰기, 기타 조작을 수행하는 저준위함수에 의존한다. 앞에서는 서로 다른 파일체계가 조작을 어떻게 처리하는지 간단히 설명하였다. 여기서는 실제장치에 대해 핵심부가 어떻게 조작을 수행하는지 고찰한다.

1. 입출력구성방식

콤퓨터가 제대로 동작하게 하려면 CPU(하나 또는 여러개), RAM 그리고 개인용콤퓨터에 련결할수 있는 여러 입출력장치사이에 정보가 흘러갈수 있도록 자료경로(data path)를 제공해야 한다. 자료경로를 통털어 모선(bus)이라고 하며 콤퓨터내부에서 가장 중요한 통신통로로 동작한다. 최근에는 ISA, EISA, PCI, MCA 등 여러 모선류형을 사용한다. 여기서는 특수한 모선류형을 자세히 설명하지 않고 모든 PC구성방식에서 공통적으로 사용하는 기능의 특징을 살펴본다. 사실 일반적으로 모선이라고 하는것은 다음 3개 종류의 규정된 모선으로 구성된다.

자료모선(data bus)

장치를 병렬로 전송하는 선로묶음이다. 펜티움은 폭이 64bit인 자료모선을 가지고있다.

주소모선(address bus)

주소를 병렬로 전송하는 행묶음이다. 펜티움은 폭이 32bit인 주소모선을 가지고있다.

조종모선(control bus)

련결된 회로에 조종정보를 전송하는 선로묶음이다. 례를 들어 펜티움은 모선을 프로쎄스와 주기억사이의 자료전송에 사용하든가 혹은 프로쎄스와 입출력장치사이의 자료전

송에 사용하는가를 지정하기 위해 조종선로를 사용한다. 읽기전송 또는 쓰기전송을 실행해야 하는가를 나타내는데도 조종선로를 사용한다.

CPU와 입출력장치의 련결을 위해 모선을 사용할 때 이 모선을 입출력모선(I/O bus)이라고 부른다. 이 경우에 Intel 80x86극소형처리기는 입출력장치주소를 나타내기 위해 주소선 32개중에서 16개를 사용하고 자료를 전송하기 위해 자료선 64개중 8, 16 또는 32개를 사용한다. 입출력모선은 3개 항목(입출력포구, 대면부, 장치조종)을 포함하는 하드웨어구성요소 계층에 의해 각 입출력장치에 련결된다. 그림 5-6은 입출력구성방식의 구성요소를 보여준다.

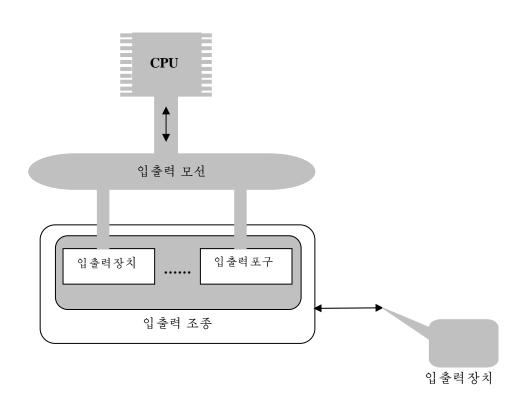



그림 5-6. PC의 입출력구성방식

1) 입출력포구

장치를 입출력모선에 련결하면 각 장치는 자기의 입출력주소모임을 소유하게 된다. 이것을 입출력포구라고 한다. IBM PC구조에서는 입출력주소공간에 8bit입출력포구가 65536개까지 들어간다. 련속되는 8bit포구 두개가 16bit포구 하나를 구성하고 16bit포구는 짝수주소에서 시작한다. 련속되는 16bit포구 두개가 32bit포구 하나를 구성하고 4의 배수주소에서 시작한다. CPU는 특수아쎔블리명령인 in, ins, out, outs를 사용하

여 입출력포구에서 값을 읽거나 입출력포구에 값을 쓸수 있다. CPU는 이 명령어중 하나를 실행하는 과정에서 주소모선을 사용하여 필요한 입출력포구를 선택하고 장치모선을 사용하여 CPU등록기와 포구사이에 장치를 전송한다.

입출력포구를 물리주소공간의 주소로 넘기기할수도 있다. 이렇게 함으로써 프로쎄스는 기억기에 직접 작용하는 아쎔블러명령어를 사용하여 입출력장치와 통신할수 있다.(례를 들면 move, and, or 등) 최근의 하드웨어장치는 기억기에 넘기기한 입출력이 더 빠르고 DMA를 사용할수 있어서 더 주목된다.

체계설계자의 중요한 목표중 하나는 입출력프로그람작성을 위한 단일접근방법을 성능저하없이 제공하는것이다. 이를 위해 각 장치의 입출력포구는 그림 5-7처럼 특수등록기모임으로 구조화되여있다. CPU는 장치에 보낼 명령을 조종등록기에 기록하고 상태등록기에서 장치의 내부상태를 나타내는 값을 읽는다. CPU는 자료를 얻어내려고 입력등록기에서 바이트를 읽으며 자료를 내보내려고 출력등록기에 값을 쓴다.

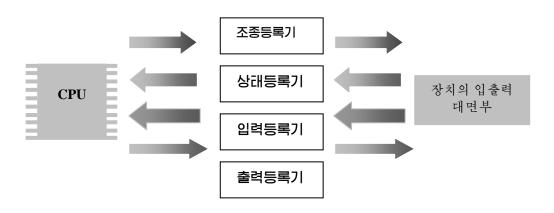



그림 5-7. 특수입출력포구

비용절약을 위해 한 입출력포구를 여러 목적으로 사용하기도 한다. 례를 들면 몇 비트는 장치상태를 나타내고 나머지 비트는 장치에 보낼 명령을 지정한다. 같은 입출력포구를 입력등록기나 출력등록기로 사용할수도 있다.

2) 입출력포구접근

입출력포구에 접근하기 위해 in, out, ins, outs 아쎔블러명령어를 사용한다. 핵심부에 포함된 다음보조함수를 사용하여 포구에 쉽게 접근할수 있다.

inb(), inw(), inl()

입출력포구에서 각각 련속되는 1,2,4B를 읽는다. 접미사 《b》. 《w》, 《l》은 각각 바이트, 단어, 배단어를 의미한다.

inb_p(), inw_p(), inl_p()

입출력포구에서 각각 련속하는 1, 2, 4B를 읽은 다음 잠시 멈추기 위해 묶음명령어

를 실행한다.

Outb(), Outw(), Outl()

각각 련속되는 1, 2, 4B를 입출력포구에 쓴다.

Outb_p(), Outw_p(), Outl_p()

각각 련속되는 1, 2, 4B를 입출력포구에 쓰고 나서 잠시 멈추기 위해 묶음명령어를 실행한다.

insb(), insw(), insl()

입출력포구에서 련속되는 바이트렬을 각각 1, 2, 4B 단위로 읽는다. 길이는 함수의 파라메터로 지정한다.

Outsb(), Outsw(), Outsl()

련속되는 바이트렬을 각각 1, 2, 4B단위로 입출력포구에 쓴다.

입출력포구에 접근하는것은 쉽지만 어떤 입출력포구가 입출력장치에 할당되여있는지 알아내는 일은 쉽지 않으며 ISA모선을 사용하는 체계인 경우 특히 까다롭다. 때때로 장치구동프로그람은 단지 하드웨어장치를 검출하기 위해 일부입출력포구에 값을 쓰는 경우도 있다. 그러나 만일 다른 하드웨어장치가 이미 이 입출력포구를 사용하고있다면 체계충돌이 발생할수 있다. 이런 경우를 방지하기 위해 핵심부는 자원(resource)을 사용하여 각 하드웨어장치에 할당된 입출력포구를 기억한다.

자원은 장치구동프로그람에 배타적으로 할당할수 있는 특정한 개체의 일부분이다. 우의 실례에서 자원은 입출력포구범위를 나타낸다. 각 자원관련정보는 자원자료구조에 보관된다. 표 5-11에 이 자료구조의 마당을 보여주었다. 같은 종류의 모든 자원을 나무와 류사한 자료구조에 삽입한다. 례를 들어 입출력포구주소범위를 나타내는 모든 자원은 ioport_resource를 뿌리로 하는 나무에 포함된다.

표 5-11. 자원자료구조의 마당

형	마 당	설 명
const char *	name	자원소유자를 나타냄
unsigned long	start	자원범위의 시작
unsigned long	end	자원범위의 끝
unsigned long	flags	여러 기발
Struct resource	parent	자원나무구조의 부모지적자
Struct resource	sibling	자원나무구조의의 형제지적자
Struct resource	child	자원나무구조의 첫번째 자식지적자

한 마디의 자식들은 목록을 구성한다. child마당이 목록의 첫번째 요소를 가리키고 sibling 마당이 목록의 다음마디를 가리킨다. 왜 나무를 사용하는가? 례를 들어 IDE하드디스크대면부가 사용하는 입출력포구주소가 0xf000에서 0xf00f까지라고 하자. start마당은 0xf000, end마당은 0xf00f로 설정된 자원을 나무에 삽입하고 name마당을 조종기의 이름으로 설정한다. 그러나 IDE 장치구동프로그람은 그 이상의 정보가 필요할수 있다. 즉 0xf000부터 0xf007까지는 주하드디스크에서 사용하고 0xf008부터 0xf00f까지는 종속하드디스크에서 사용한다. 이를 위해 장치구동프로그람은 0xf000부터 0xf00f 범위를 나타내는 자원의 아래에 각 부분범위(subrange)를 나타내는 자식마디 두개를 삽입한다. 일반적인 규칙으로 나무의 각 마디는 부모마디가 나타내는 범위에 속하는 부분범위를 나타내야 한다. 어떤 장치구동프로그람이든지 자원나무의 뿌리마디, 필요한 자원자료구조의 주소를 파라메터로 다음 3개 함수를 사용할수 있다.

request_resource()

주어진 범위를 입출력장치에 할당한다.

check_resource()

주어진 범위가 비여있는지, 그중 일부를 이미 입출력장치에 할당했는지 검사한다.

release_resource()

이전에 입출력장치에 할당한 범위를 해제한다.

핵심부는 이 함수들을 입출력포구에 적용한 함수를 제공한다. request_region()은 주어진 범위의 입출력포구를 할당한다. check_region()은 주어진 범위의 입출력포구가 비여있는지 또는(일부라도) 사용중인지 검사한다. release_region()은 이전에 할당한 입출력포구범위를 해제한다. 현재입출력장치에 할당한 모든 입출력주소의 나무를 /proc/ioports 파일에서 얻을수 있다.

2. 입출력대면부

입출력대면부는 입출력포구그룹과 이것들에 대응하는 장치조종기(device controller) 사이에 위치하는 하드웨어회로이다. 대면부는 입출력포구에 넣어준 값을 장치에 대한 명령과 장치로 변환하는 해석기역할을 수행하며 반대방향으로 장치의 상태변화를 감시하여 상태등록기역할을 하는 입출력포구값을 변경한다. 이 회로는 IRQ행을 통해 프로그람가능한 새치기조종기(PIC, Programmable Interrupt Controller)에 련결되여 장치를 대신해서 새치기요청을 발생시킨다.

대면부에는 두가지 류형이 있다.

o **전용입출력대면부**

한 종류의 특정한 하드웨어장치를 위한것이다. 어떤 경우 입출력대면부를 포함하는 카드자체에 장치조종기가 위치하기도 한다. 전용입출력대면부에 련결된 장치는 내부장치(PC본체내부에 있는 장치)일수도 있고 외부장치(PC본체 외부에 있는 장치)일수도 있다.

o 일반입출력대면부

여러가지 다양한 하드웨어장치를 련결할 때 사용한다. 일반입출력대면부에 련결된 장치는 언제나 외부장치이다.

1) 전용입출력대면부

전용입출력대면부가 얼마나 다양한지 알수 있도록 요즘 PC에 설치된 장치중에서 가장 일반적인것을 서술하였다.

o 건반대면부

건반대면부는 전용극소형처리기를 가지고있는 건반조종기에 련결된다. 이 극소형처리기는 사용자가 누른 건조합을 해석하고 새치기를 발생시켜 건조합에 대응하는 주사코드를 입력등록기에 넣는다.

o 도형대면부

도형대면부는 도형카드에 들어있는 대응하는 조종기와 일체형으로 되여있다. 도형카드는 자기의 프레임완충기와 특수처리기, ROM소편에 보관된 코드를 보유하고있다. 프레임완충기는 현재화면내용의 도형서술(description)을 포함하는 기억기이다.

o 디스크대면부

디스크대면부는 케블을 통해 디스크조종기(disk controller)에 련결된다. 디스크조종기는 일반적으로 디스크에 통합되여있다. 레를 들어 IDE대면부는 띠형케블 40선을 통해 디스크자체에 있는 지능적디스크조종기에 련결되여있다.

o 모선마우스대면부

모선마우스대면부는 마우스에 들어있는 대응하는 조종기와 케블을 통해 련결된다.

o 망대면부

망대면부는 망카드에 들어있는 대응하는 조종기와 일체형으로 되여있다. 망카드는 망파케트을 주고받는데 사용된다. 널리 쓰이는 몇가지 망표준이 있으나 이써네트(Ethernet)가 가장 일반적이다.

2) 일반입출력대면부

최근에 나오는 PC는 다양한 일반입출력대면부를 포함한다. 이것들은 광범한 외부장치를 련결한다. 가장 일반적인 대면부는 다음과 같다.

o 병렬포구

전통적으로 인쇄기를 련결하는데 사용한다. 외장형디스크, 화상입력장치, 일시복사장치, 다른 콤퓨터 등과 련결하는데 사용할수도 있다. 자료를 한번에 바이트(8bit)단위로 전송한다.

o 직렬포구

병렬포구와 비슷하지만 자료를 한번에 1bit씩 전송한다. 직렬포구는 UART(Universal Asynchronous Receiver and Transmitter)소편을 가지고있으며 이것을 사용하여 내보내는 바이트를 비트렬로 변환하고 들어오는 비트렬을 다시 바이트

로 재조립한다. 이 대면부는 병렬포구보다 느리므로 고속으로 동작할 필요가 없는 외부장치 즉 모뎀, 마우스, 인쇄기 등에 사용한다.

o 일반직렬모선(USB)

최신일반입출력대면부로 빠르게 확산되고있다. 고속으로 동작하며 전통적으로 병렬포구나 직렬포구에 련결하던 외부장치를 위해 사용한다.

o PCMCIA대면부

이동가능한 콤퓨터(노트형콤퓨터)에 주로 사용한다. 외부장치는 신용카드처럼 생겼는데 체계를 다시 초기시동하지 않고도 슬로트(slot)에 넣거나 슬로트에서 제거할수 있다. 가장 일반적인 PCMCIA장치는 하드디스크, 모뎀, 망카드 RAM확장 등이다.

o SCSI대면부

주PC모선을 SCSI(Small Computer System Interface)모선이라는 보조모선에 련결하는 회로이다. SCSI-2모선은 PC와 외부장치(하드디스크, 화상입력장치, CD-ROM쓰기장치 등)를 8개까지 련결할수 있다. 광역SCSI-2와 SCSI-3대면부는 장치를 16개까지 허용하며 대면부를 추가로 사용할 경우 더 확장할수 있다. SCSI표준은 SCSI모선을 통해 장치를 련결하기 위한 통신규약이다.

3. 장치조종기

복잡한 장치를 구동하기 위해 장치조종기(dcvice controller)가 필요할수도 있다. 조종기는 두가지 중요한 역할을 수행한다.

입출력대면부에서 받은 고수준명령을 해석하고 적절한 전기신호를 차례로 장치에 보내서 장치가 특수한 작업을 수행하도록 한다. 장치에서 받은 전기신호를 변환하고 적절히 해석한 다음 (입출력대면부를 통해)상태등록기값을 변경한다.

전형적인 장치조종기로 디스크조종기를 들수 있다. 디스크조종기는 극소형처리기로부터(입출력대면부를 통해) 《이 블로크의 장치를 기록하라.》 같은 고수준명령을 받아서 《디스크자두를 옳바른 자리길로 옮기라》, 《자료를 자리길에 기록하라》 같은 저준위디스크연산으로 변환한다. 최신디스크조종기는 매우 복잡해서 디스크자료를 고속기억기캐쉬에 보관하거나 받은 고수준명령의 순서를 바꿔서 실제 디스크배치에 최적화되도록 할수도 있다.

더 단순한 장치에는 장치조종기가 없다. 프로그람가능한 새치기조종기(《새치기와 례외》참조)와 프로그람가능한 간격시간(《프로그람가능한 간격시간》참조) 등이 그 실례이다.

여러 하드웨어장치가 독자적인 기억기를 보유하고있는데 이것들을 입출력공유기억기(I/0 shared memory)라고 부른다. 례를 들어 모든 최신도형카드는 프레임완충기라는 수MB에서 수십MB이상인 RAM이 있으며 모니터에 출력할 화면영상를 보관하는데 사용한다.

1) 입출력공유기억기주소의 사영

장치와 모선류형에 따라 PC구성방식의 입출력공유기억기는 3가지 다른 물리주소범위와 대응한다.

ISA모선에 련결된 대부분의 장치

입출력공유기억기는 일반적으로 물리주소범위 0xa0000에서 0xfffff에 대응한다.

이 범위는 《예약된 페지틀》에서 언급한 640kB에서 1MB 사이의 《구멍》에 해당한다.

VESA국부모선을 사용하는 오래된 장치

VESA는 도형카드에서 주로 사용하는 특수한 모선으로서 입출력공유기억기는 주소범위 0xe00000에서 0xffffff에 대응한다. 즉 14MB에서 16MB 범위다. 이런 장치는 페지화표의 초기화를 더 복잡하게 만들며 더는 생산되지 않는다.

PCI모선에 련결된 장치

입출력공유기억기는 RAM의 물리주소범위를 훨씬 지나서 아주 큰 물리주소에 대응한다. 이 장치는 아주 다루기 쉽다. 최근 Intel은 고성능도형카드를 위해 PCI를 향상한 AGP(Accelerated Graphics Port)표준을 도입하였다. 이 카드는 독자적인 입출력공유기억기가 있을뿐아니라 GART(Graphics Address Remapping Table)라는 특수하드웨어회로를 통해 주기판의 RAM령역을 직접 참조할수 있다. GART회로를 사용하여 AGP카드는 구형PCI카드보다 훨씬 고속으로 자료를 전송할수 있다. 그러나 핵심부관점에서 물리적인 기억기가 어디에 위치하는지는 문제되지 않으며 GART배치된 기억기도 다른 종류의 입출력공유기억기와 같은 방법으로 처리한다.

2) 입출력공유기억기접근

핵심부는 어떻게 입출력공유기억기의 특수한 위치에 접근하는가? 일단 다루기 쉬운 PC구성방식부터 시작하고 이어서 다른 하드웨어구성방식을 고찰하자.

핵심부프로그람이 선형주소를 사용하여 동작한다는 사실은 특수한 입출력공유기억기위치도 PAGE_OFFSET보다 큰 주소로 표현해야 한다. 여기서는 PAGE_OFFSET이 0xc0000000이라고 가정한다. 즉 핵심부선형주소가 3GB부터 4GB사이에 있다. 핵심부구동프로그람은 반드시 특수한 입출력공유기억기위치의 입출력물리주소를 핵심부공간의 선형주소로 변환해야 한다. PC구성방식에서는 단순히 32bit 물리주소를 상수 0xc0000000과 OR연산을 수행하면 된다. 례를 들어 핵심부가 t1에 물리적주소 0x000b0fe4인 입출력위치의 값을 보관하고 t2에 물리적주소 0xfc000000인 입출력위치의 값을 보관해야 한다고 하자.

그렇다면 다음과 같이 하면 된다고 생각할수도 있다.

```
t1=*((unsigned char*)(0xc00b0fe4));
t2=*((unsigned char*)(0xfc000000));
```

초기화과정에서 핵심부는 RAM의 물리적주소를 선형주소공간 3GB-4GB의 시작부분에 대응시킨다. 따라서 페지화유니트는 첫번째 문장의 0xc00b0fe4선형주소를 원래

입출력물리주소 0x000b0fe4에 대응시키며 이 주소는 640kB에서 1kB사이의 《ISA구역》부분에 해당한다.(4장 1절의 《Linux페지화》 참조)

그러나 두번째 문장의 경우 문제가 발생한다. 입출력물리주소가 체계RAM의 마지막물리주소보다 크기때문이다. 따라서 0xfc000000 선형주소가 반드시 0xfc000000물리주소와 대응할 필요는 없다. 이런 경우 입출력물리주소에 대응하는 선형주소를 포함하도록 핵심부페지표을 바꾸어야 한다. 이것은 ioremap()함수 또는 ioremap_nocache()를 호출하여 처리할수 있다.

이 함수는 vmalloc()과 류사하게 get_vm_area()를 호출하여 새로운 vm_struct 서술자(4장 2절의 《불련속적인 기억기령역서술자》 참고)를 생성한다. 이 서술자는 요청한 입출력공유기억기령역만큼의 크기를 가진 선형주소범위를 나타낸다. 그리고 나서 ioremap()함수는 기본핵심부페지표에서 해당하는 페지표항목을 적당히 갱신한다. ioremap_nocache()는 ioremap()와 달리 다시 배치한 선형주소를 참조할 때 하드웨어캐쉬를 비활성화한다.(include\asm-i386\io.h)

따라서 두번째 문장의 옳바른 형식은 다음과 같이 될것이다.

```
io_mem = ioremap(0xfb000000, 0x200000);
t2=*((unsigned char *)(io_mem + 0x100000));
```

첫번째 문장은 0xfb000000에서 시작하는 새로운 2MB의 선형주소범위를 생성한다. 두번째 문장은 0xfc000000주소를 가지는 기억기위치를 읽는다. 나중에 배치를 제거하려면 장치구동프로그람이 iounmap()를 사용해야 한다. PC이외의 하드웨어구성방식중에서 어떤 경우에는 물리적기억기위치를 가리키는 선형주소를 통해 입출력공유기억기주소에 접근할수 없다. 따라서 Linux는 다음과 같은 구성방식고유의 마크로를 정의하며 입출력공유기억기에 접근할 때 반드시 이것들을 사용해야 한다.

readb, readw, readl

입출력공유기억기위치에서 각각 1, 2, 4B를 읽는다.

writeb, writew, writel

입출력공유기억기위치에 각각 1, 2, 4B를 쓴다.

memcpy_fromio, memcpy_toio

입출력공유기억기위치에서 동적기억기로 혹은 그 반대로 장치블로크를 복사한다.

memset_io

입출력공유기억기령역을 특수한 값으로 채운다.

0xfc000000 입출력위치에 접근하기 위해 권고할만한 방법은 다음과 같다.

```
io_mem = ioremap(0xfb000000, 0x200000);
t2 = readb(io_mem + 0x100000);
```

이 마크로덕분에 입출력공유기억기에 접근할 때 특수한 기반에 대한 모든 의존성을 감출수 있다.

4. DMA

모든 PC에는 직접기억기접근조종기(Direct Memory Access Controller), 줄여서 DMAC라는 보조처리기가 있다. CPU는 RAM과 입출력장치사이에서 자료를 전송하도록 이 프로쎄스에 명령을 내릴수 있다. CPU가 DMAC를 활성화하고 나면 DMAC는 스스로 자료전송을 계속할수 있다. DMAC는 자료전송이 끝나면 새치기요청을 발생시킨다. CPU와DMAC가 동시에 같은 기억기위치에 접근하려 할 때 발생하는 충돌은 기억기중재자(memory arbiter)라는 하드웨어회로가 해결한다.

많은 량의 자료를 한번에 전송하는 디스크구동프로그람과 같은 느린 장치가 DMAC를 주로 사용한다. DMAC의 초기설정시간이 상대적으로 길기때문에 전송할 자료의 량이 적은 경우는 CPU를 직접 사용하여 자료를 전송하는것이 더 효과적이다. 오래된 ISA모선에서 사용한 초기DMAC는 복잡하고 프로그람작성하기도 어려웠으며 물리적기억기의 시작부분에 있는 16MB만을 사용할수 있었다. PCI나 SCSI모선을 위한 최신 DMAC는 모선에 있는 전용하드웨어회로에 의존하며 장치구동프로그람개발자를 훨씬 편하게 해주었다.

지금까지 세 종류의 기억기주소를 보았다. 즉 CPU가 내부적으로 사용하는 론리주소와 선형주소, CPU가 자료모선을 물리적으로 구동하기 위해 사용하는 기억기주소인 물리적주소이다. 마지막으로 4번째 기억기주소인 모선주소가 있다. 이것은 CPU를 제외한 모든 하드웨어장치가 자료모선을 구동하기 위해 사용하는 기억기주소이다. PC구성방식에서 모선주소는 물리주소와 동일하지만 SUN의 SPARC, HP의 Alpha와 같은 구성방식에서는 이것들 주소가 서로 다르다.

핵심부가 왜 모선주소에 주의를 돌려야 하는가? DMA연산에서 자료전송은 CPU의 간섭없이 이루어지며 입출력장치와 DMAC자료모선을 직접 구동한다. 따라서 핵심부가 DMA연산을 설정할 때 핵심부는 사용할 기억기완충기의 모선주소를 DMAC 또는 입출력장치의 적절한 입출력포구에 기록해야 한다.

많은 입출력구동프로그람이 연산속도를 높이려고 DMAC를 활용한다. DMAC는 자료전송을 위해 장치의 입출력조종기와 호상작용하며 핵심부는 DMAC프로그람작성을 위해 쉽게 사용할수 있는 루틴을 제공한다. 입출력조종기는 자료전송을 완료하면 IRQ를 통해 CPU에 신호를 보낸다.

장치구동프로그람이 입출력장치에 대해 DMA연산을 설정하는 경우 모선주소를 사용하여 사용할 기억기완충기를 지정해야 한다. 핵심부는 선형주소를 모선주소로 변환하기 위한 virt_to_bus 그리고 그 반대로 변환하기 위한 bus_to_virt마크로를 제공한다.(include\asm-i386\io.h)

IRQ행과 마찬가지로 DMAC도 요청하는 구동프로그람에 동적으로 할당해야 하는 자원이다. 구동프로그람이 DMA 연산을 시작하고 끝내는 방법은 모선류형에 따라 다르

다. PCI, SCSI와 같은 모선의 경우 IRQ행의 할당과 DMA전송시동이라는 2단계로 실행된다. 장치파일을 열 때 DMA 연산의 완료를 알리기 위해 사용할 IRQ행을 할당한다.(《장치구동프로그람초기화》참고) DMA연산을 시작하기 위해 장치구동프로그람은 DMA완충기의 모선주소, 전송방향, 자료의 크기를 하드웨어장치의 입출력포구에 기록한다. 구동프로그람은 현재프로쎄스를 대기시킨다. DMA전송이 끝나면 하드웨어장치가 새치기를 발생시켜 장치구동프로그람을 깨운다. 마지막프로쎄스가 파일객체를 닫으면 장치파일의 release메쏘드가 IRQ행을 해제한다.

5. 장치파일

1장에서 고찰한바와 같이 Unix계렬조작체계는 파일이라는 개념에 바탕을 둔다. 파일은 바이트의 련속으로 이루어진 정보를 보관하는 그릇이다. 이 접근방법에 따라 입출력장치도 파일로 취급한다. 따라서 디스크에 있는 정규파일을 읽고쓸 때 사용하는 체계호출을 입출력장치에도 사용할수 있다. 례를 들어 같은 write()체계호출을 사용하여 정규파일에 자료를 쓰기도 하고 /dev/lp0 장치파일에 써서 인쇄기로 보낸다.

기초로 되는 장치구동프로그람의 특성에 따라 장치파일을 블로크(block)와 문자(character)라는 두 류형으로 분류한다. 두가지 하드웨어장치의 차이는 명확하지는 않지만 최소한 다음과 같이 말할수 있다.

- 블로크장치의 자료는 임의로 참조할수 있다. 자료블로크를 전송하는데 걸리는 시간이(최소한 사람이 보기에) 짧으며 항상 거의 같다. 전형적인 블로크장치의 실례는 하드디스크, 유연성디스크, CD-ROM, DVD재생기 등이다.
- 문자장치의 자료는 임의로 참조할수 없거나(례를 들면 음성카드) 임의로 참조할수는 있지만 임의의 위치의 자료에 접근하는 시간은 장치안에서의 위치에 크게 의존한다.(례를 들면 자기테프구동프로그람)

망카드는 이 류형분류에서 대표적인 례외이며 파일에 직접 대응하지 않는 하드웨어장치이다.

Linux에는 체계의 등록부나무에 실제파일이 보관되는 구식(old-style)장치파일과 /proc파일체계처럼 가상파일인 devfs장치파일이라는 두 종류의 장치파일이 있다. 그러면 이 두 종류의 장치파일을 더 자세히 보자.

1) 구식장치파일

구식장치파일은 Unix조작체계 초기판본부터 사용되여왔다. 구식장치파일은 파일체계에 보관된 실제파일이다. 그러나 파일의 색인마디는 디스크의 자료블로크를 참조하지 않는다. 대신 색인마디는 하드웨어장치를 나타내는 식별자를 가지고있다. 장치이름과 류형(앞에서 설명한것처럼 블로크인지 문자인지)외에 각 장치파일은 중요한 속성 두개를 가지고있다.

주번호(Major number)

장치류형을 나타내는 1에서 254사이의 수자이다. 일반적으로 동일한 주번호와 류형을 가진 모든 장치파일은 같은 장치구동프로그람에 의해 처리되며 따라서 동일한 파일연산모임을 공유한다.

주번호(Minor number)

같은 주번호를 공유하는 여러 장치중에서 특수한 장치를 나타내는 수자이다.

구식장치파일을 생성하기 위해 mknod() 체계호출을 사용한다. 이 체계호출의 매개변수는 장치파일의 이름과 류형 그리고 주, 부번호이다. 주, 부번호라는 두 파라메터를 16bit dev_t 수자로 변환한다. 여기서 웃자리 8bit는 주번호, 아래자리 8bit는 부번호를 나타낸다. MAJOR, MINOR마크로를 사용하여 16bit수자에서 두 값을 추출할수 있으며 MKDEV마크로를 사용하여 주번호와 부번호를 합쳐서 16bit 수자로 만든다. 실제로 dev_t는 응용프로그람에서 주로 사용하는 자료형이고 핵심부는 kdev_t 자료형을 사용한다. 현재의 Linux핵심부에서 이 두 형은 unsigned short int지만 미래의 Linux판본에서 kdev_t는 완전한 장치파일서술자가 될것이다. 주번호와 부번호는 색인마디객체의 i_rdev마당에 보관한다. 장치파일의 류형(블로크인지 문자인지)은 i_mode마당에 보관한다.

일반적으로 장치파일은 /dev등록부에 보관한다. 표 5-12에서 몇가지 장치파일의 속성을 보여준다.

일반적으로 장치파일은 하드웨어장치(/dev/hda 같은 하드디스크) 또는 하드웨어장치의 물리적 또는 론리적인 일부(/dev/hda2 같은 디스크구획)와 대응한다. 그러나 어떤 경우에 장치파일은 실제하드웨어장치에 대응하지 않고 가상적인 론리적장치를 나타낸다. 례를 들어 /dev/null은 《검은 구멍》과 같은 장치파일이다. 이 장치파일에 기록되는 모든 자료는 사라지며 파일은 언제나 비여있는것처럼 보인다.

표 5-12. 장치파일실례

이름	류형	주번호	부번호	설 명
/dev/fd0	블로크	2	0	유연성디스크
/dev/had	블로크	3	0	첫번째 IDE디스크
/dev/hda2	블로크	3	2	첫번째 IDE디스크의 두번째 기본구획
/dev/hdb	블로크	3	64	두번째 IDE디스크
/dev/hdb3	블로크	3	67	두번째 IDE디스크의 세번째 기본구획
/dev/ttyp0	문자	3	0	말단
/dev/console	문자	5	1	콘솔

/dev/lp1	문자	6	1	병렬인쇄기
/dev/ttyso	문자	4	64	첫번째 직렬포구
/dev/rtc	문자	10	135	실시간박자
/dev/null	문자	1	3	빈장치(검은구멍)

2) Devfs장치파일

장치파일을 주번호와 부번호를 사용하여 표현하는데는 몇가지 한계가 있다.

1. /dev등록부에 있는 장치의 대부분이 실제로 존재하지 않는다. 새로운 입출력장치를 설치할 때마다 체계관리자가 새로운 장치파일을 설치할 필요가 없도록 /dev등록부에 장치파일들이 있다. 그러나 보통 /dev등록부에 장치파일이 1800개이상 있으며 처음 참조할 때 색인마디탐색시간을 증가시킨다.

2. 주번호와 부번호는 각각 8bit이다. 이 값은 어떤 하드웨어장치의 경우 한계가 된다. 례를 들어 초대형체계에 있는 SCSI장치를 식별할 때 문제가 생긴다.(Linux에서는 우회적인 방법으로 SCSI 디스크구동프로그람에 여러 주번호를 할당하였다. 그 결과 핵심부는 SCSI 디스크를 128개까지 지원한다.)

devfs장치파일은 이런 문제들과 다른 사소한 문제를 해결하기 위해 도입되였다. 그러나 아직까지도 광범하게 사용되고있지 않다. 따라서 코드를 설명하지 않고 주요개념만 설명한다.

devfs가상파일체계는 구동프로그람이 주번호, 부번호 대신 이름을 사용하여 장치를 등록하도록 한다. 핵심부는 특수한장치를 탐색하는 작업을 쉽게 하려고 기본명명규칙을 제공한다. 실례를 들어 모든 디스크장치는 /dev/discs가상등록부에 있다. 따라서 /dev/hda는 /dev/discs/disc

0, /dev/hdb는 /dev/discs/disc1이 될것이다. 사용자는 장치관리데몬을 적절히 설정하여 이전의 이름규칙도 계속 사용할수 있다. evfs 파일체계를 사용하는 입출력구동프로그람은 devfs_register()를 호출하여 장치를등록한다. 함수는 장치파일이름, 장치구동프로그람메쏘드표을 가리키는 지적자를 포함하는 새로운 devfs_entry구조체를 생성한다.

등록된 장치파일은 자동으로 devfs가상등록부에 나타난다. 이 등록부에 있는 장치파일의 색인마디객체는 파일에 접근할 때에만 생성된다. devfs파일의 덴트리객체가 파일연산을 가리키는 지적자를 가지고있기때문에 장치파일을 여는것이 좀 더 효률적이다.(뒤부분에 있는 《장치구동프로그람초기화》참고)

그러나 devfs에도 몇가지 문제가 있다. 가장 중요한 점은 주번호, 부번호는 Unix 체계에서 어느 정도 불가피하다는 사실이다. 첫째로 NFS봉사기나 find명령과 같은 사용자방식응용프로그람은 주어진 파일을 포함하는 물리적디스크구획을 나타내기 위해 주번호, 부번호에 의존한다. 둘째로 장치번호는 POSIX표준에서도 요구하고있다. 따라서

devfs계층은 핵심부가 구식장치파일처럼 각 장치구동프로그람에 대해 주번호와 부번호를 정의하도록 하고있다. 현재 거의 모든 장치구동프로그람은 대응하는 구식장치파일과 같은 주번호, 부번호를 devfs장치파일에 련관시키고있다. 이런 리유로 여기에서도 계속 구식장치파일에 주로 중점을 둔다.

3) VFS의 장치파일처리

장치파일은 체계등록부안에 있지만 정규파일이나 등록부와 근본적으로 다르다. 프로쎄스가 정규파일에 접근하는 경우 파일체계를 통해 디스크구획내의 자료블로크에 접근한다. 그러나 프로쎄스가 장치파일에 접근하는 경우 프로쎄스는 (파일의 내용을 읽거나 쓰지 않고) 단지 하드웨어장치를 구동할뿐이다. 례를 들어 프로쎄스가 콤퓨터에 련결된 수자형온도계로부터 방의 온도를 알아내려고 장치파일을 사용할수 있다. 응용프로그람이 장치파일과 정규파일을 구별하지 못하도록 하는것은 VFS책임이다.

이것을 처리하기 위해 장치파일을 열 때 VFS가 장치파일의 기본파일연산을 변경한다. 그 결과 해당 장치파일에 대한 체계호출은 장치파일이 들어있는 파일체계가 제공하는 함수가 아닌 장치와 관련한 함수의 호출로 바뀐다. 장치와 관련한 함수는 하드웨어장치에 작용하여 프로쎄스가 요청한 연산을 실행한다.

프로쎄스가(블로크형이든 문자형이든)장치파일에 대해 open()체계호출을 실행하였다고 하자. 요약하면 체계호출에 대응하는 봉사루틴이 경로명을 장치파일로 판단하고 파일에 대응하는 색인마디객체, 덴트리객체, 파일객체를 생성한다. 구식장치파일의 경우를 보면 파일체계의 적절한 함수(보통 ext2_read_inode())를 사용하여 파일에 대응하는 색인마디를 디스크에서 읽어서 색인마디객체를 초기화한다. 디스크색인마디가 장치파일의것이라고 판단하면 init_special_inode()를 호출하여 색인마디객체의 i_rdev마당을 장치파일의 주번호, 부번호로 초기화하고 색인마디객체의 i_fop 마당을 장치파일의 류형에 따라 def_blk_fops구조체 또는 def_chr_fops구조체로 설정한다. open()체계호출의 봉사루틴은 dentry_open()함수를 호출하며 이 함수는 새로운 파일객체를 할당하고 파일객체의 f_op마당을 i_fop에 들어있는 주소로 설정한다.

이 주소는(장치파일의 형에 따라) def_blk_fops 또는 fef_chr_fops의 주소이다. 이 두 구조체의 내용은 뒤에 나오는 《블로크장치구동프로그람》과 《문자장치구동프로그람》에서 볼수 있다. 이것들을 통해서 장치파일을 대상으로 호출한 체계호출은 파일체계함수대신 장치구동프로그람함수를 호출하게 된다.

6. 장치구동프로그람

장치구동프로그람은 잘 정의된 프로그람작성대면부를 통해 하드웨어장치에 물어보고 대답하게 해주는 쏘프트웨어계층이다. 이 대면부는 이미 익숙한 함수들이다. 즉 VFS 함수(open, read, lseek, ioctl 등)를 장치조종에도 사용한다. 장치구동프로그람이 이 함수들을 실제로 구현한다. 각 장치는 독자적인 입출력조종기(I/O controller)를 가지

고있으며 따라서 독자적인 명령과 상태정보를 포함하기때문에 대부분의 입출력장치에는 독자적인 장치가 있다.

장치구동프로그람에는 많은 류형이 있다. 이것들은 사용자방식응용프로그람에 제공하는 지원수준, 하드웨어장치에서 수집한 자료를 완충하는 기법 등의 측면에서 크게 다르다. 이런 측면이 장치구동프로그람의 내부구조에 큰 영향을 주기때문에 이에 관해서는 《핵심부지원수준》과 《장치구동프로그람의 완충기법》에서 설명한다.

장치구동프로그람은 장치파일연산을 구현하는 함수만으로 구성되지 않는다. 장치구동프로그람을 사용하기 전에 장치구동프로그람의 등록과 초기화라는 두가지 일을 수행해야 한다. 또한 장치구동프로그람이 자료를 전송할 때 입출력연산을 감시(monitor)해야 한다. 《장치구동프로그람등록》과 《장치구동프로그람초기화》, 《입출력연산의 감시》에서 이런 일을 어떻게 처리하는지 고찰할것이다.

1) 핵심부지원수준

Linux핵심부는 존재하는 모든 입출력장치를 완벽하게 지원하지는 않는다. 간단히 말해서 하드웨어장치에 대한 지원에는 다음과 같은 3종류가 있다.

o **지원하지 않음**

응용프로그람이 적절한 in, out아셈블러명령을 사용해서 장치의 입출력포구와 직접 호상작용한다.

o **최소지원**

핵심부는 하드웨어장치를 인식하지 않고 단지 해당 입출력대면부만 인식한다. 사용자프로그람은 해당 대면부를 일련의 문자렬을 읽고 쓸수 있는 순차적장치로 다룰수 있다.

o **확장지원**

핵심부는 하드웨어장치를 인식하고 입출력대면부를 직접 처리한다. 사실 장치에 대한 장치파일조차 없는 경우도 있다.

첫번째 접근방법 즉 핵심부의 장치구동프로그람에 의존하지 않는 접근방법의 례로 X윈도우체계가 도형표시장치를 처리하는 방법을 들수 있다. 이 접근방법은 매우 효률적이지만 입출력장치가 발생시키는 하드웨어새치기를 X 봉사기가 활용할수 없다는 부족점이 있다. 또 필요한 입출력포구에 접근하기 위해 추가적인 노력이 필요하다. 3장 1절의 《과제상태토막》에서 취급한것처럼 iopl()과 ioperm() 체계호출을 사용하여 프로쎄스에 입출력포구에 접근할수 있는 특권을 줄수 있다. 뿌리(root)권한을 소유한 프로그람만이 체계호출을 실행할수 있지만 실행파일의 fsuid마당을 초사용자(8장 4절의 《프로쎄스의 자격과 특질》참고)의 UID를 0으로 설정하면 일반사용자도 프로그람을 사용할수 있다.

최신Linux판본은 사용자들이 많이 사용하는 대부분의 도형카드를 지원한다. /dev/fb장치파일이 도형카드의 프레임완충기에 대한 추상화를 제공하여 응용프로그람 쏘프트웨어가 도형대면부의 입출력포구에 관해 전혀 알 필요없이 도형카드에 접근할수

있도록 한다. 뿐만아니라 Linux핵심부는 DRI(Direct Rendering Infrastructure)를 지원하여 응용프로그람쏘프트웨어가 3D가속도형카드의 하드웨어를 활용할수 있게 해준다. 그렇지만 아직도 직접 모든 일을 처리하는 X윈도우봉사기도 많이 사용되고있다.

최소지원방법은 일반입출력대면부에 련결된 외부하드웨어장치를 처리하는데 쓰인다. 핵심부는 장치파일(따라서 결국 장치구동프로그람)을 제공하여 입출력대면부를 관리한다. 응용프로그람은 장치파일을 읽고 쓰는것으로 외부하드웨어장치를 처리한다.

핵심부크기를 작게 유지할수 있기때문에 확장지원방법보다 최소지원방법을 선택한다. 그렇지만 PC에서 볼수 있는 일반입출력대면부에서는 단지 직렬포구와 병렬포구만 이 접근방법을 사용한다. 따라서 직렬마우스는 X봉사기 같은 응용프로그람으로 직접 조종하며 직렬모뎀에는 항상 Minicom, Seyon과 같은 통신프로그람이나 PPP(Point to Point Protocol)데몬이 필요하다.

외부장치가 핵심부내부자료구조와 밀접하게 호상작용하는 경우에는 최소지원방법을 사용할수 없기때문에 제한된 범위의 응용프로그람만 사용할수 있다. 례를 들어 외장형 하드디스크가 일반입출력대면부에 련결되여있다고 가정하자. 응용프로그람은 디스크를 인식하고 디스크의 파일체계를 탑재하는데 필요한 모든 핵심부자료구조, 함수와 호상작용할수 없으므로 이 경우 확장지원방법을 사용해야 한다.

일반적으로 내부하드디스크처럼 입출력모선에 직접 련결된 모든 하드웨어장치는 확장지원접근방법에 따라 처리한다. 핵심부는 각 장치에 대해 장치구동프로그람을 제공해야 한다.

USB, 노트콤에서 볼수 있는 PCMCIA포구, SCSI대면부 등 간단히 말해 직렬포구와 병렬포구 이외의 모든 일반입출력대면부와 련결하는 외부장치는 모두 확장지원방법을 사용한다.

open(), read(), write() 같은 파일관련표준체계호출이 언제나 기반하드웨어장치의 모든 조종권을 응용프로그람에 주지는 않는다는 점을 기억해야 한다. 사실 최소공통기능만을 제공하는 VFS의 접근방법은 어떤 장치가 특수한 내부상태에 있는지 여부를 응용프로그람이 알수 있도록 하는 특수명령을 제공할 여지가 없다.

그런 필요를 만족시키기 위해 POSIX표준에서 ioctl()체계호출을 도입하였다. 이 체계호출은 장치파일의 파일서술자요청을 지정하는 32bit 파라메터외에 임의로 추가적인 파라메터를 가질수 있다. 례를 들어 CD-ROM의 음성크기를 얻거나 CD-ROM디스크를 배출하는 특수한 ioctl()요청이 존재한다. 응용프로그람은 이런 ioctl()요청을 사용하여 CD재생기와 같은 사용자대면부를 구현할수 있다.

2) 장치구동프로그람의 완충기법

전통적으로 Unix계렬의 조작체계는 하드웨어장치를 블로크와 문자장치로 구분한다.

그러나 이 분류가 모든것을 말해주는것은 아니다. 어떤 장치는 입출력연산 한번으로 많은 량의 자료를 전송할수 있는 반면에 다른 장치는 단지 문자 몇개만을 전송하기때문

이다. 례를 들어 PS/2마우스구동프로그람은 읽기연산 한번으로 몇 바이트만을 읽는다. 이것들은 마우스단추의 상태와 화면에서 마우스지적자의 위치 등에 대응한다. 이런 장치는 가장 다루기 쉬운 종류에 속한다. 한번에 한 문자씩 장치의 입력등록기에서 입력자료를 읽어서 적절한 핵심부자료구조에 보관한다. 출력자료도 프로쎄스주소공간에서 핵심부자료구조로 복사한 다음 한번에 하나씩 입출력장치의 출력등록기에 쓴다. 이런 장치의 경우 DMA입출력연산을 설정하는 시간이 입출력포구에서 자료를 읽거나 쓰는데 걸리는 시간과 비슷할 정도이기때문에 입출력구동프로그람은 DMAC를 사용하지 않는다.

반면에 핵심부는 각 입출력연산에서 많은량의 자료를 쏟아내는 장치도 처리할수 있어야 한다. 이런 장치에는 음성카드, 망카드 같은 순차적장치나 모든 종류(유연성, CDROM, SCSI 디스크)의 디스크 같은 임의의 접근장치들이 포함된다.

례를 들어 콤퓨터에 들어있는 음성카드를 설정해서 마이크에서 들어오는 소리를 록음하려 한다고 가정하자. 음성카드는 마이크에서 들어오는 전기적인 신호를 44.14kHz의 고정된 주기로 표본화해서 입력자료로부터 련속적인 16bit수값을 생성한다. 음성카드구동프로그람은 어떤 경우라도 심지어 CPU가 다른 프로쎄스를 실행하느라 바쁜중에도 이 많은 자료를 처리할수 있어야 한다.

이 문제를 해결하기 위해 두가지 기법을 결합하여 사용한다.

- 자료블로크전송을 위해 DMA프로쎄스를 사용한다.
- 각 요소가 자료블로크크기인 요소 두개 이상을 가지는 원형완충기를 사용한다.

새로운 자료블로크를 읽었음을 알리는 새치기가 발생하면 새치기처리기는 원형완충기의 요소를 가리키는 지적자를 증가시켜 빈 요소에 자료를 보관할수 있도록 한다.

반대로 구동프로그람이 자료블로크를 사용자주소공간으로 성공적으로 복사하면 원형완충기의 요소를 해제하여 하드웨어장치에서 오는 새로운 자료를 보관하는데 사용하도록 한다. 원형완충기는 CPU부하의 최고값(peak)을 낮추는 역활을 한다. 자료를 받아들이는 사용자방식응용프로그람이 우선순위가 높은 다른 과제때문에 늦어지더라도 새치기처리기는 현재실행프로쎄스중에 실행되므로 DMAC는 원형완충기의 요소를 계속 채울수 있다.

망카드에서 파케트를 받을 때에도 비슷한 상황이 발생한다. 이 경우 들어오는 자료의 흐름은 비동기적이다. 파케트는 서로 독립적으로 수신되며 린접하는 두 파케트의 도착시간간격을 예측할수 없다.

그럼에도 같은 완충기를 재사용하지 않으므로 순차적장치의 완충기처리는 쉬운 편이다. 음성응용프로그람은 마이크에 같은 자료블로크를 다시 전송해달라고 요청할수 없다. 망응용프로그람도 망카드에 같은 파케트을 다시 전송해달라고 요청할수 없다.

반면에 임의의 접근장치(모든 종류의 디스크)에 대한 완충은 훨씬 더 복잡하다. 이 경우 응용프로그람은 같은 자료블로크를 반복해서 읽거나 쓸수 있다. 더군다나 이런 장치에 대한 접근은 매우 느리다. 이런 이상한 특성이 디스크구동프로그람의 구조에 큰 영

향을 준다. 따라서 임의의 접근장치의 완충기는 다른 역활을 수행한다. CPU부하의 최고값을 낮추는 대신 어떤 프로쎄스에서도 당장은 더 필요없는 자료를 가지고있으면서 다른 프로쎄스가 나중에 완충기안에 들어있는 자료를 요청하는 경우를 대비한다. 다시 말해서 완충기는 디스크접근의 회수를 줄이는 쏘프트웨어캐쉬(4장 3절 참고)의 기본구성요소이다.

3) 장치구동프로그람등록

앞에서 핵심부가 장치파일에 대한 체계호출을 대응하는 장치구동프로그람의 적절한 함수호출로 변환한다는 사실을 보았다. 이를 위해 장치구동프로그람은 자기를 등록해야 한다. 다시 말해서 장치구동프로그람을 등록한다는것은 장치구동프로그람과 대응하는 장치파일과 련결한다는것을 의미한다. 구동프로그람이 아직 등록되지 않은 장치파일에 접근하면 오유코드 ENODEV를 반환한다.

장치구동프로그람이 정적으로 콤파일되여 핵심부에 포함되여있다면 핵심부초기화과정에서 구동프로그람의 등록이 이루어진다. 장치구동프로그람이 핵심부모듈로 콤파일되여있다면 모듈을 적재할 때 등록이 이루어진다. 모듈의 경우 장치구동프로그람은 모듈이 탑재해제될 때 자기의 등록을 취소할수도 있다. 구식장치파일을 사용하는 문자장치구동프로그람을 나타내는 자료구조는 char_device_struct의 배렬인 chrdevs이다.

```
static struct char_device_struct {
    struct char_device_struct *next;
    unsigned int major;
    unsigned int baseminor;
    int minorct;
    const char *name;
    struct file_operations *fops;
    struct cdev *cdev;          /* will die */
} *chrdevs[MAX_PROBE_HASH];
(\fs\char_dev.c)
struct cdev {
    struct kobject kobj;
    struct module *owner;
    struct file_operations *ops;
    struct list_head list;
    dev_t dev;
    unsigned int count;
};
(\include\linux\cdev.h)
```

각 배렬첨수는 장치파일의 주번호이다. 주번호의 범위는 1(주번호 0을 가질수있는 장치파일은 없다)에서 254(255는 이후확장을 위해 예약되여있다.)까지이므로 배렬은 요소를 255개 포함한다. 그렇지만 배렬의 첫번째 요소는 사용되지 않는다.

각 char_device_struct구조체는 name, fops라는 두 마당을 포함한다. name은 장치클라스의 이름을 가리키고 fops는 file_operations구조체를 가리킨다. 이와 류사하게 블로크장치구동프로그람은 자료구조 255개의 배렬인 blkdevs로 나타낸다.(chrdevs 배렬과 마찬가지로 첫번째 요소는 사용되지 않는다.) 각 구조체는 다음 두 마당을 포함한다. name은 장치클라스의 이름을 가리키고 bdops는 block_device_operations구조체를 가리킨다. bdops에는 블로크장치구동프로그람의 핵심적인 연산을 위한 몇가지 독자적인 메쏘드가 있다.(표 5-13참고)

표 5-13. 블로크장치구동프로그람의 메쏘드

메쏘드	메쏘드의 실행을 시작하도록 하는 사건
Open	블로크장치파일을 여는 경우
release	블로크장치파일이 마지막 참조를 닫는 경우
ioctl	블로크장치파일에 대해 idcd()체계호출을 실행하는 경우
check_media_change	매체가 바뀌었는지 검사한다. (례를 들면 유연성디스크)
revalidatc	블로크장치가 유효한 자료를 가지고있는지 검사한다.

chrdevs와 blkdevs배렬은 초기에는 비여있다. register_chrdev()와 register_blkdev()함수를 사용하여 이 배렬에 새로운 요소를 삽입한다.(include\linux\fs.h, fs\char_dev.c, fs\block_dev.c)

모듈로 구현된 장치구동프로그람의 경우 모듈이 탑재해제될 때 unregister_chrdev()함수를 사용하여 등록을 취소할수 있다.

례를 들어 병렬인쇄기구동프로그람 클라스를 위한 서술자를 chrdevs배렬에 다음과 같이 삽입할수 있다.

register_ chrdev(6, "lp", &lp_fops);

첫번째 파라메터는 주번호를 나타내고 두번째는 장치클라스이름, 마지막은 파일연산의 배렬을 가리킨다. 등록하고나면 장치구동프로그람이 장치파일의 경로명이 아닌 장치파일의 주번호와 련결된다. 따라서 경로명과 관계없이 장치파일에 대한 접근은 대응하는 구동프로그람을 활용한다.

4) 장치구동프로그람초기화

장치구동프로그람의 등록과 초기화는 아주 다른 개념이다. 사용자방식응용프로그람

이 장치파일을 통해 사용할수 있도록 장치구동프로그람의 등록은 최대한 빨리 이루어지는 반면에 초기화는 최대한 늦게 이루어진다. 사실 구동프로그람을 초기화하면 체계의 비싼 자원을 할당하게 되며 다른 구동프로그람이 사용할수 없게 된다.

《입출력새치기처리》에서 본것처럼 IRQ의 경우에도 여러 장치가 한 IRQ행을 공유할수 있도록 IRQ를 장치에 할당하는것은 사용직전에 동적으로 이루어진다, 가능하면 늦게 할당할수 있는 자원으로 DMA전송완충기 DMA통로를 위한 페지틀을 들수 있다.(유연성디스크구동프로그람과 같은 비PCI 장치의 경우)

장치구동프로그람은 필요할 때 자원을 얻을수 있게 하면서도 이미 할당된 자원을 여러번 요청하는것을 막으려고 다음과 같은 기법을 사용한다.

사용계수기가 현재장치파일에 접근하고있는 프로쎄스의 수를 관리한다. 계수기는 장치파일의 open메쏘드에서 증가하고 release메쏘드에서 감소한다.

open메쏘드는 사용계수기의 값을 검사한 다음 증가시킨다. 계수기가 0이면 장치구동프로그람이 자원을 할당하고 하드웨어장치에 대해 새치기와 DMA를 활성화해야 한다.

release메쏘드는 감소시킨 다음 계수기의 값을 검사한다. 계수기가 0이면 하드웨어장치를 사용하는 프로쎄스가 없다. 이 경우 메쏘드는 입출력조종기에 대해 새치기와 DMA를 비활성화하고 할당된 자원을 해제한다.

7. 입출력연산조종

입출력연산의 지속시간은 예측하기 어렵다. 기계적인 고려사항(전송할 블로크에 대한 디스크자두의 현재위치)은 말그대로 임의적인 사건(자료파케트가 언제 망카드에 도착할것인가) 또한 인위적인 요소(사용자가 언제 건반을 누를것인지 또는 언제 인쇄기에 종이가 걸렸는지 알아차릴것인가)에 따라 달라진다. 어떤 경우든 입출력연산을 시작한 장치구동프로그람은 입출력연산을 완료했는지 또는 시간넘침이 되였는지 알려주는 조종기법에 의존해야 한다.

연산을 완료하면 장치구동프로그람은 입출력대면부의 상태등록기를 읽어서 입출력연산을 제대로 실행했는지 결정한다. 시간넘침의 경우 해당 연산에 주어진 시간이 지났고 아무런 일도 발생하지 않았으므로 무엇인가 잘못되였다는 사실을 구동프로그람이 알게 된다. 입출력연산의 완료를 조종하는 기법으로 문의방식(polling mode)과 새치기방식(interrupt mode)이 있다.

1) 문의방식

이 기법을 사용하면 입출력연산을 마쳤다는 의미로 상태등록기의 값이 변경될 때까지 CPU장치의 상태등록기를 계속 반복하여 검사(문의)한다. 프로쎄스가 이미 사용중인 스핀잠그기를 얻으려 할 때 그 값이 0이 될 때까지 계속 반복하여 검사한다. 그러나 입출력연산에 문의을 적용하면 기다리는 시간이 너무 길고 구동프로그람의 시간넘침시간도 기억해야 하므로 비용이 훨씬 많이 들게 된다.

간단한 문의의 례는 다음과 같다.

처리기의 시간랑비를 줄이기 위해 장치구동프로그람은 각 문의연산이 끝난 다음에 자발적으로 CPU를 놓아주어 다른 실행가능한 프로쎄스를 계속 실행할수 있도록 한다.

```
for (;;){
    if (read_status(device) &DEVICE END_OPERATION)  break;
    if (--count == 0) break;
}
```

순환에 들어가기 전에 초기화한 count변수는 반복할 때마다 감소하므로 단순한 시간넘침기구를 구현하는데 사용할수 있다. 더 정확한 시간넘침기구를 위해 반복할 때마다 틱크계수기 jiffies값을 읽어서 대기순환(wait loop)을 시작하기 전에 읽은 값과 비교하여 구현할수 있다.

입출력연산을 완료할 때까지 소요한 시간이 상대적으로 길어서 수ms이상이라면 이 기법은 입출력이 완료하기를 기다리며 CPU가 귀중한 실행시간을 랑비하게 하므로 비효률적이다. 따라서 매 문의연산 다음에 schedule()함수호출을 삽입하여 자발적으로 CPU를 풀어주는것이 좋을것이다.

2) 새치기방식

새치기방식은 입출력조종기가 IRQ행을 통해 입출력연산이 끝났음을 신호로 알려줄수 있을 때에만 사용할수 있다. 단순한 경우 새치기방식이 어떻게 동작하는지 보자. 단순한 입력문자장치를 위한 구동프로그람을 작성한다고 가정하자. 사용자가 장치에 대응하는 장치파일에 대해 read()체계호출을 실행하면 입력명령이 장치의 조종등록기로 전송된다. 긴 시간간격이 지난 다음 장치가 자료 한 바이트를 입력등록기에 넣는다. 장치구동프로그람은 read()체계호출의 결과로 그 바이트를 반환한다.

이런 경우가 새치기방식으로 구동프로그람을 구현하는것이 좋은 전형적인 경우이다. 사실 장치구동프로그람은 하드웨어장치에서 결과를 받으려고 얼마나 오래동안 기다려야하는지 알수 없다. 핵심적으로 구동프로그람은 두 함수를 포함한다.

1. foo_read()함수는 파일객체의 read메쏘드를 구현한다.

2. foo_interrupt()함수는 새치기를 처리한다.

foo_read()함수는 사용자가 장치파일을 읽을 때 구동된다.

장치구동프로그람은 foo_dev_t형의 독자적인 서술자를 사용한다. 서술자는 하드웨어장치에 대한 동시접근을 방지해주는 신호기 sem, 대기렬 wait, 장치가 새치기를 발생시키면 설정되는 기발 intr, 새치기가 쓰고 read메쏘드가 읽는 한 바이트완충기 data 등을 포함한다. 일반적으로 새치기를 사용하는 모든 입출력구동프로그람은 새치기처리기와 read, write메쏘드가 같이 접근하는 자료구조를 사용한다. foo_dev_t서술자의 주소는 일반적으로 장치파일의 파일객체에 있는 private_data마당이나 대역변수에 보관된다.

foo_read()함수의 주요동작은 다음과 같다.

1. foo_dev->sem 신호기를 획득하여 다른 프로쎄스가 장치에 접근하지 못하도록 한다.

2. intr기발을 지운다.

3. 입출력장치에 read명령을 보낸다.

4. wait_event_interruptible을 실행하여 intr기발이 1이 될 때까지 프로쎄스를 연기한다.

시간이 지난 다음 장치는 새치기를 발생시켜 입출력연산을 완료했으며 자료가 적절한 DEV_FOO_DATA_PORT 자료포구에 있다는 사실을 알린다. 새치기처리기는 intr기발을 설정하고 프로쎄스를 깨운다. 순서짜기프로그람이 프로쎄스를 재실행하도록 결정한 경우 foo_read()의 나머지 부분을 다음과 같이 실행한다.

1.foo_dev->data변수에 들어있는 문자를 사용자주소공간으로 복사한다.

2.foo_dev->sem신호기를 해제한 다음 완료한다.

간단하게 설명하려고 시간넘침조종에 관해서는 설명하지 않았다. 일반적으로 시간넘침조종은 정적 또는 동적시계를 사용하여 구현한다. 시계는 입출력연산시작전에 설정하고 연산을 완료하면 제거한다.

이제 foo_interrupt()함수를 보자.

```
void foo_interrupt (int irq, void *dev_ id,struct pt_regs *regs)
{
        foo->data =inb(DEV_FOO_DATA_PORT);
        foo->intr=1;
        wake_up_interruptible(&foo->wait);
}
```

새치기처리기는 장치의 입력등록기에서 문자를 읽어서 foo대역변수가 가리키는 장치구동프로그람의 서술자 foo_dev_t에 있는 data마당에 보관한다. 다음으로 intr기발을 설정하고 wake_up_interruptible()을 호출하여 foo->wait대기렬에 차단되여있는 프로쎄스를 깨운다.

새치기처리기에서 3개의 파라메터중 하나도 사용하지 않았는데 이것이 일반적인 경우이다.

8. 블로크장치구동프로그람

하드디스크와 같은 전형적인 블로크장치는 접근시간이 아주 길다. 하드디스크조종기가 디스크표면우에 있는 자두를 자료가 있는 위치로 정확하게 옮겨야 하기때문에 보통 각 연산을 완료할 때까지 수ms가 걸린다. 그러나 자두이동이 끝나면 자료전송은 초당 수십MB속도로 진행된다.

하드디스크나 그와 류사한 장치는 충분한 성능을 얻기 위해 린접한 여러 바이트를 한번에 전송한다. 앞으로는 디스크표면에 있는 탐색연산 한번으로 접근가능한 여러 바이트를 가리켜 《바이트가 린접한다.》는 표현을 사용할것이다.

Linux블로크장치운전기의 구조는 아주 복잡하다. 여기서는 블로크장치운전기를 지원하려는 목적으로 핵심부에 포함된 모든 함수를 자세히 설명하지는 않지만 전체적인 쏘프트웨어구성방식을 보고 주요자료구조를 소개한다. 블로크장치운전기를 위해 핵심부는 다음과 같은 특징을 지원한다.

- VFS를 통해 단일한 대면부제공
- 디스크장치에 대한 효률적인 미리읽기(read-ahead)구현
- 자료에 대한 디스크캐쉬제공

1) 블로크장치구동프로그람관리

블로크장치파일을 열려고 할 때 핵심부는 장치파일이 이미 열려있는지 확인해야 한다. 파일이 이미 열려있다면 핵심부는 대응하는 블로크장치구동프로그람을 초기화하면 안된다.

그러나 이 문제는 보기처럼 쉽지 않다. 앞에서 장치파일에서 주번호가 같은 블로크장치파일이 같은 블로크장치구동프로그람에 대응한다고 설명하였다. 그러나 부번호 여러개를 처리하는 블로크장치구동프로그람은 개별적인 블로크장치구동프로그람 여러개를 생각할수 있으므로 문제되지 않는다. 앞으로 여기에서 블로크장치구동프로그람이라는 용어는 주번호 하나와 부번호 하나로 지정한 하드웨어장치에 대한 입출력을 처리하는 핵심부계층을 의미한다.

문제를 복잡하게 만드는 점은 VFS가 주번호와 부번호는 같지만 경로명이 다른 블로크장치파일을 서로 다른 파일로 생각하지만 사실은 같은 블로크장치구동프로그람을 가리킨다는 사실이다. 따라서 핵심부는 색인마디캐쉬에서 블로크장치파일에 대한 객체가 존재하는지 검사하는것만으로는 블로크장치구동프로그람이 이미 사용중인지 판단할수 없다.

어떤 블로크장치구동프로그람이 현재 사용중인지 추적하기 위해 핵심부는 주번호와 부번호 조합을 통해 참조하는 하쉬표을 사용한다. 핵심부는 블로크장치구동프로그람을 사용할 때마다 주번호와 부번호 조합으로 나타내는 블로크장치구동프로그람이 이미 하쉬표에 보관되여있는지 검사한다. 표에 있다면 블로크장치구동프로그람은 이미 사용중이다. 블로크장치파일의 주번호와 부번호에 대해 하쉬함수를 사용하므로 주어진 블로크장치파일을 통해 접근할 때 블로크장치구동프로그람이 활성화되였는지 아니면 주번호와 부번호가 같은 다른 블로크장치파일을 통해 접근했는지 하는것은 상관없다. 주번호와 부번호 조합에 대응하는 블로크장치구동프로그람을 찾을수 없으면 핵심부는 새로운 요소를 하쉬표에 추가한다. 하쉬표배렬은 bdevhashtable배렬에 보관되는데 배렬은 블로크장치서술자의 목록 64개를 포함한다. 각 서술자는 block_device자료구조이며 마당은 표 5-14와 같다.

(include\linux\fs.h)

표 5-14. 블로크장치서술자의 마당

형	마 당	설 명
struct list_head	bd_ hash	하쉬표목록지시자
atomic_t	bd_conut	블로크장치서술자의 사용계수기
struct inode *	bd_inode	블로크장치구동프로그람의 주색인마디객체를 가리키는 지적자
Dev_t	bd_dev	블로크장치의 주번호와 부번호
Int	bd_openers	블로크장치구동프로그람이 열린 회수
struct block_device_ operations*	bd_op	블로크장치구동프로그람연산표의 지시자
struct semaphore	bd_sem	블로크장치구동프로그람을 보호하는 신호기
struct list_head	bd_inodes	이 구동프로그람을 사용해서 연 블로크장치파일의 색인마디목록

블로크장치서술자의 bd_inodes 마당은 색인마디의 2중련결원형목록의 머리부(첫번째 묶음요소)를 보관한다. 색인마디는 이 블로크장치구동프로그람을 사용해서 열린 블로크장치파일을 나타낸다. 색인마디객체의 i_devices마당이 이 목록의 앞, 뒤요소를 가리킨다.

각 블로크장치서술자는 bd_inode마당에 구동프로그람의 특수블로크장치색인마디객체의 주소를 보관한다. 이 색인마디는 디스크파일에 대응하지 않고 bdev특수파일체계에 속한다. 블로크장치색인마디는 같은 블로크장치를 가리키는 블로크장치파일의 색인마디객체가 공유하는 정보의 《원본》을 가지고있다.

2) 블로크장치구동프로그람의 초기화

이제 블로크장치구동프로그람을 어떻게 초기화하는지 보자. 《VFS의 장치파일 처리》에서 블로크장치파일을 열 때 핵심부가 어떻게 파일객체의 메쏘드를 변경하는지 보았다. f_op마당을 def_blk_fops변수의 주소로 설정한다. 이 표의 내용은 표 5-15에 있다. dentry_open()함수에서 open 메쏘드가 정의되여있는지 검사한다. open은 블로크장치파일의 경우에 언제나 정의되여있으며 blkdev_open() 함수를 실행한다.

다음의 구조체는 블로크장치구동을 위한 기본함수들에 대한 대면이라고 볼수 있다.

```
struct file_operations def_blk_fops = {
    .open       = blkdev_open,
    .release    = blkdev_close,
```

```
    .llseek             = block_llseek,
    .read          = generic_file_read,
    .write         = blkdev_file_write,
    .aio_read      = generic_file_aio_read,
    .aio_write     = blkdev_file_aio_write,
    .mmap          = generic_file_mmap,
    .fsync         = block_fsync,
    .ioctl         = block_ioctl,
    .readv         = generic_file_readv,
    .writev             = generic_file_write_nolock,
    .sendfile      = generic_file_sendfile,
};
```

표 5-15.　　블로크장치파일의 기본파일연산메쏘드

메쏘드	블로크장치파일인 경우 해당하는 함수
open	blkdev_open()
release	blkdev_close()
llseek	block_llseek()
read	generic_file_read()
write	generic_file_write()
mmap	generic_file-mmap()
fsync	block_fsync()
ioctl	blkdev_ioctl()

bd_acquire() 함수는 다음연산을 실행한다.

1. 색인마디객체에 대응하는 블로크장치파일이 이미 열려있는지 검사한다.(열린 경우 inode->i_bdev마당은 블로크장치서술자를 가리킨다.)

파일이 이미 열려있으면 블로크장치서술자의 사용계수기(inode->i_bdev->bd_count)를 증가시키고 되돌이한다.

2. inode->rdev에 보관된 주번호와 부번호를 사용하여 하쉬표에서 블로크장치구동프로그람을 탐색한다. 구동프로그람이 사용중이 아니여서 서술자를 찾지 못했으면 블로

크장치에 대해 새로운 block_device 새로운 색인마디객체를 할당하고 새로운 서술자를 하쉬표에 삽입한다.

3. 블로크장치구동프로그람서술자의 주소를 inode->i_bdev에 보관한다.

4. inode를 구동프로그람서술자의 색인마디목록에 삽입한다.

다음으로 blkdev_open()이 do_open()을 호출하여 다음과 같은 단계를 실행한다.

1. 블로크장치구동프로그람서술자의 bd_op 마당이 NULL이면 이 마당을 블로크장치파일의 주번호에 대응하는 blkdevs표의 요소로 초기화한다.

2. 블로크장치구동프로그람서술자의 open메쏘드(bd_op->open)가 정의되여있다면 이것을 실행한다.

3. 블로크장치구동프로그람서술자의 bd_openers 계수기를 증가시킨다.

4. 블로크장치색인마디객체(bd_inode)의 i_size, i_blkbits마당을 설정한다.

블로크장치구동프로그람서술자의 open메쏘드는 블로크장치구동프로그람의 메쏘드를 더 설정할수 있으며 자원을 할당하고 블로크장치파일의 부번호에 따라 다른 처리를 할수 있다.

장치구동프로그람초기화함수는 반드시 장치파일에 대응하는 물리적블로크장치의 크기를 판단해야 한다. 이 길이를 blk_size대역배렬에 장치파일의 주번호와 부번호를 색인으로 하여 1kB 단위로 보관한다.

3) 분구, 블로크, 완충기

블로크장치에 대한 자료전송연산은 《분구(sector)》라는 련속되는 바이트그룹을 단위로 수행한다. 대개 디스크장치의 분구크기는 512B이다. 좀더 큰 분구(1024, 2048B)를 사용하는 장치도 있다. 반드시 자료전송의 기본단위인 분구를 기억하여야 한다. 디스크장치가 여러 린접분구를 한번에 전송할수 있지만 한 분구보다 작은 크기는 전송할수 없다.

핵심부는 각 하드웨어블로크장치의 분구크기를 hardsect_size라는 표에 보관한다.(\include\linux\blkdev.h의 request_queue구조체의 unsigned short형으로 선언되여있다.)

표의 각 요소는 대응하는 블로크장치파일의 주, 부번호를 통해 참조한다. 따라서 hardsect_size[3][2] 는 /dev/hda2, 즉 첫번째 IDE디스크내에 있는 두번째 구획의 분구크기를 나타낸다.(표 5-12참고) hardsect_size[maj]이 NULL인 경우 주번호 maj를 공유하는 모든 블로크장치는 표준분구크기인 512B를 가진다. 블로크장치구동프로그람은 《블로크(block)》라는 많은 련속바이트를 한번에 전송한다. 블로크를 분구와 혼돈하지 말아야 한다. 분구는 하드웨어장치에 대한 자료전송의 기본단위지만 블로크는 장치구동프로그람이 요청하는 단일입출력연산과 관련한 련속바이트그룹이다.

Linux에서 블로크크기는 반드시 2의 제곱수여야 하고 최대크기는 페지틀의 크기이다. 그리고 블로크에는 분구가 정수개 있기때문에 반드시 분구크기의 배수여야 한다. 따

라서 PC구성방식에서 허용되는 블로크크기는 512, 1024, 2048, 4096B이다. 블로크장치구동프로그람 하나가 같은 주번호를 공유하는 여러 장치파일을 처리해야 하고 각 블로크장치파일이 자기의 블로크크기를 가질수 있기때문에 블로크장치구동프로그람은 여러 블로크크기에 동작할수 있어야 한다. 례를 들어 어떤 블로크장치구동프로그람이 Ext2파일체계와 교환령역을 담은 두 구획이 있는 하드디스크를 처리해야 한다고 하자. 이 경우 장치구동프로그람은 Ext2구획의 경우 1024B, 교환구획의 경우 4096B를 사용한다.

핵심부는 blksize_size라는 표에 블로크크기를 보관한다. 표의 각 요소는 대응하는 블로크장치파일의 주, 부번호로 참조한다. blksize_size[maj]이 NULL이면 주번호 maj를 공유하는 모든 블로크장치의 블로크크기는 표준블로크크기인 1024B이다.(blk_size를 blksize_size배렬과 혼돈하지 말자. blksize_size는 블로크장치자체의 크기가 아닌 블로크장치의 블로크크기를 보관한다.)

각 블로크는 자기의 완충기 즉 핵심부가 블로크의 내용을 보관하기 위해 사용하는 RAM기억기령역을 요청한다. 장치구동프로그람이 디스크에서 블로크를 읽을 때 하드웨어장치에서 얻은 값을 사용하여 해당 완충기를 채운다. 류사하게 장치구동프로그람이 디스크에 블로크를 기록할 때 완충기의 실제값을 사용하여 하드웨어장치의 련속바이트를 갱신한다. 완충기크기는 언제나 대응하는 블로크크기와 일치한다.

4) 완충기머리부

완충기머리부는 각 완충기와 관련한 buffer_head형서술자이다. 여기에는 핵심부가 완충기를 다루는데 필요한 모든 정보가 들어있다. 따라서 핵심부는 각 완충기를 처리하기 전에 완충기머리부를 검사한다.

표 5-16에 완충기머리부의 마당을 서술하였다. 각 완충기머리부의 b_data마당에는 해당 완충기의 시작주소를 보관한다. 페지를 하나가 여러 완충기를 보관할수 있으므로 b_this_page 마당이 페지내 다음완충기의 완충기머리부를 가리킨다. 이 마당은 전체 페지틀을 보관하고 검색하는데 사용된다. b_blocknr마당은 론리적블로크번호 즉 디스크구획내 블로크색인을 보관한다.

표 5-16. 완충기머리부 마당

형	마 당	설 명
struct buffer_head *	b_next	충돌하쉬목록의 다음항목
unsigned long	b_blocknr	론리적블로크번호
unsigned short	b_size	블로크크기
kdev_t	b_dev	가상장치식별자
atomic_t	b_count	블로크사용계수기

kdev_t	b_rdev	실제장치식별자
unsigned long	b_state	완충기상태기발
unsigned long	b_flushtime	완충기플래시(flash)시간
struct buffer_head *	b_next_free	목록의 다음항목
struct buffer_head*	b_prev_free	목록의 이전항목
struct buffer_head*	b_this_page	페지별 완충기목록
struct buffer_head*	b_reqnext	요청대기렬의 다음항목
struct buffer_head **	b_pprev	충돌하쉬목록의 이전항목
char*	b_data	완충기를 가리키는 지시자
struct page*	b_page	완충기를 보관하고있는 페지의 서술자지시자
void(*)()	b_end_io	입출력완료메쏘드
void(*)	b_private	장치구동프로그람의 개별자료
unsigned long	b_rsector	실제장치안에서의 블로크번호
wait_qucue_head_t	b_wait	완충기대기렬
struct inode *	b_inode	완충기가 속한 색인마디객체를 가리키는 지시자
struct list_head	B_inode_buffers	색인마디완충기목록의 지시자

b_state 마당은 다음기발을 보관한다.

BH_Uptodate

완충기에 유효한 자료가 있으면 1로 설정한다. buffer_uptodate()마크로가 이 기발값을 반환한다.

BH_Dirty

완충기가 불결하면 즉 블로크장치에 기록해야 하는 자료가 있으면 1로 설정한다. buffer_dirty()마크로가 이 기발값을 반환한다.

BH_ Lock

완충기에 잠그기가 걸려있으면 1로 설정한다.디스크전송에 사용하는 완충기에서 이런 경우가 발생한다. buffer_locked()마크로가 이 기발값을 반환한다.

BH_Req

대응하는 블로크를 요청했으며 유효한(최신)자료를 포함하고있으면 1로 설정한다. buffer_req()마크로가 이 기발값을 반환한다.

BH_Mapped

완충기가 디스크에 배치되여있으면 즉 대응하는 완충기머리부의 b_dev, b_blocknr마당이 유효하면 1로 설정한다. buffer_mapped()마크로가 이 기발값을 반환한다.

BH_New

대응하는 파일블로크가 방금 할당되여 아직 한번도 접근되지 않았으면 1로 설정한다. buffer_new()마크로가 이 기발값을 반환한다.

BH_Async

end_buffer_io_async()가 처리중인 완충기이면 1로 설정한다. buffer_async()마크로가 이 기발값을 반환한다.

BH_ Wait_IO

기억기를 회수할 때 완충기청소를 연기하는데 사용한다.

BH_ launder

기억기를 회수할 때 완충기가 청소되는중일 때 설정한다.

BH_ JBD

기록형파일체계가 사용하는 완충기일 때 설정한다.

b_dev마당이 완충기에 보관된 블로크를 포함하는 가상(virtual)장치를 나타내는 반면에 b_rdev마당은 실제(real)장치를 나타낸다. 간단한 일반하드디스크의 경우 이 구분이 의미가 없지만 동시에 동작하는 디스크 여러개로 이루어진 RAID(Redundant Array of Independent Disks)기억기를 나타내기 위해 도입하였다. 안전성과 효률성을 높이기 위해 RAID배렬에 보관한 파일은 여러 디스크에 나누어 보관하지만 응용프로그람은 론리디스크 하나로 생각한다. 따라서 b_blocknr 마당은 론리적인 블로크번호를 나타내고 b_rdev마당은 특수한 디스크장치를 나타내며 b_rsector마당은 분구번호를 나타낸다.

5) 블로크장치구동프로그람구성방식의 개요

블로크장치구동프로그람은 한번에 한 블로크를 전송할수 있지만 핵심부는 디스크에 요청한 각 블로크에 대해 매번 별도의 입출력연산을 수행하지 않는다. 디스크표면에서 블로크의 물리적인 위치를 찾는 시간이 매우 오래 걸리기때문에 매번 별도의 입출력연산을 수행하면 디스크성능이 매우 나빠지기때문이다. 대신 핵심부는 될수 있으면 여러 블로크를 한번에 처리하여 자두의 평균이동회수를 줄인다.

프로쎄스VFS계층 또는 다른 핵심부구성요소가 디스크블로크를 읽거나 쓰려고 하면 실제로는 《블로크장치요청》을 생성한다. 이것은 요청된 블로크와 블로크에 실행할 연

산(읽기 또는 쓰기)을 나타낸다. 그렇지만 핵심부는 요청이 들어오는 즉시 처리하지 않는다. 이 입출력연산은 단지 실행예정목록에 추가되고 나중에 실행한다. 이렇게 하는것이 과제를 뒤로 연기하는것처럼 보이지만 블로크장치성능을 향상하는 핵심적인 기구이다. 핵심부가 새로운 블로크장치자료전송을 요청받으면 대기중인 이전의 요청을 약간 확대하여 이 요청을 만족시킬수 있는지 검사한다. 즉 탐색(seek)연산을 하지 않고도 새로운 요청을 만족시킬수 있는지 검사한다. 디스크를 순차적으로 접근하는 경우가 많기때문에 이 간단한 기구는 매우 효률적이다.

요청을 연기하는것은 블로크장치처리를 매우 복잡하게 한다. 례를 들어 어떤 프로쎄스가 정규파일 하나를 열고 이어서 파일체계구동프로그람이 파일에 대응하는 색인마디를 디스크에서 읽으려 한다고 가정하자. 블로크장치구동프로그람은 요청을 대기렬에 넣고 색인마디를 가지고있는 블로크가 전송될 때까지 프로쎄스를 보류한다. 그러나 블로크장치구동프로그람자체는 차단상태가 될수 없다. 그렇게 되면 같은 디스크에 접근하려는 다른 모든 프로쎄스도 차단으로 되기때문이다.

블로크장치구동프로그람의 보류를 막기 위해 각 입출력요청을 비동기적으로 처리한다. 따라서 자료전송을 완료할 때까지 대기상태가 될수 있는 핵심부조종경로는 없다. 특히 블로크장치구동프로그람은 새치기구동방식이기때문에 《고준위구동프로그람》은 새로운 블로크장치요청을 생성하거나 기존의 블로크장치요청을 확대한 다음 즉시 실행을 완료할수 있다. 나중에 활성화되는 《저준위구동프로그람》은 소위 《전략루틴(strategy routine)》을 호출한다. 이 루틴은 대기렬에서 요청을 받아 디스크조종기에 적절한 명령을 지시하여 요청을 처리한다. 입출력연산을 완료하면 디스크조종기는 새치기를 발생시키고 해당 운전기는 전략루틴을 다시 실행한다. 필요하다면 렬에 있는 다른 요청도 처리한다. 각 블로크장치구동프로그람은 자기의 《요청대기렬(request queue)》이 있다. 물리적블로크장치마다 요청대기렬이 하나씩 있어야 하는데 이렇게 함으로써 요청순서를 정렬하여 디스크성능을 높일수 있다. 전략루틴은 순차적으로 렬을 탐색하여 자두이동을 최소로 하여 모든 요청을 처리한다.

○ **요청서술자**

요청서술자(request descriptor)는 각 블로크장치요청을 표현한다. 요청서술자는 표 5-17과 같은 request자료구조에 보관한다. 자료전송방향은 cmd마당에 보관된다. 이 마당의 값은 READ(블로크장치에서 RAM으로) 또는 WRITE(RAM에서 블로크장치로)이다. 요청상태를 지정하기 위해 rq_status마당을 사용한다.

대부분의 블로크장치에서 RQ_INACTIVE(사용하지 않는 요청서술자) 또는 RQ_ACTIVE(저준위장치구동프로그람에 의해 이미 처리되였거나 처리예정인 유효한 요청)이다.

표 5-17. 요청서술자마당

형	마 당	설 명
struct list_head	queue	요청대기렬목록의 지시자
int	elevator_sequence	승강기알고리듬을 위한 요청의 나이 (age)[a]
volatile int	rq_staus	요청상태
kdev_t	rq_dev	장치식별자
int	cmd	요청한 연산
int	errors	성공 또는 실패코드
unsigned long	sector	(가상) 블로크장치의 첫번째 분구번호
unsigned long	nr_sector	(가상) 블로크장치에 요청한 분구수
unsigned long	hard_sector	(실제) 블로크장치의 첫번째 분구번호
unsigned long	hard_nr_sectors	(실제) 블로크장치에 요청한 분구수
unsigned int	nr_segments	(가상) 블로크장치에 요청한 토막수
unsigned int	nr_hw_segments	(실제) 블로크장치에 요청한 토막수
unsigned long	current_nr_sectors	현재 전송중인 블로크분구수
void *	special	SCSI장치의 구동프로그람에서 사용함
char*	buffer	입출력전송을 위한 기억기령역
struct complction*	waiting	요청에 관련된 대기렬
struct buffer_head*	bh	요청의 첫번째 완충기서술자
struct buffer_head*	bhtail	요청의 마지막 완충기서술자
request_queue_t*	q	요청대기렬서술자지시자

요청은 동일한 장치의 린접하는 여러 블로크를 포함할수 있다. rq_dev마당은 블로크장치를 나타내고 sector마당은 요청의 첫번째 블로크에 있는 첫번째 분구번호를 나타낸다. nr_sector는 요청에 있지만 아직 처리하지 않은 분구의 수를 나타낸다.

current_nr_sector는 요청의 첫번째 블로크에 있는 분구수를 나타낸다.

뒤에서 보지만 sector, nr_sector, current_nr_sector마당은 요청을 처리하는 동안 동적으로 바뀔수 있다. nr_segments마당은 요청의 토막수를 나타낸다. 요청한 모든 블로크는 블로크장치에서는 반드시 린접해야 하지만 완충기는 RAM에서 반드시 련속일 필요는 없다. 요청에서 린접하고 대응하는 완충기들도 RAM에서 련속인 블로크들을 《토막(segment)》라고 부른다. 저준위장치구동프로그람은 한 토막의 모든 블로크를 연산 한번으로 전송하도록 DMA조종기를 프로그람작성할수 있다.

hard_sector, hard-nr_sectors, nr_hw_segments 마당은 보통 각각 sector, nr_sectors, nr_segments와 값이 같다. 그러나 요청이 여러 물리적블로크장치를 한번에 처리하는 구동프로그람을 가리키는 경우 값이 서로 다르다. 이와 같은 구동프로그람의 전형적인 례는 LVM(론리볼륨관리자)이다.

LVM은 여러 디스크를 관리하며 여러 디스크구획을 가상디스크구획 하나처럼 다룬다. 이 경우 hard_가 붙은 마당은 실제물리적블로크장치를 가리키고 붙지 않은 마당은 가상장치를 가리키므로 서로 다르다. 또 다른 례는 쏘프트웨어RAID이다. 이 구동프로그람은 자료를 여러 디스크에 복제하여 안정성을 높인다.

요청내 모든 블로크의 완충기머리부는 단순련결목록을 구성한다. 각 완충기머리부의 b_reqnext마당은 목록의 다음요소를 가리키고 요청서술자의 bh와 bhtail마당은 각각 목록의 처음과 마지막요소를 가리킨다. 요청서술자의 buffer마당은 실제자료전송에 사용하는 기억기령역을 가리킨다.

블로크 하나를 요청하는 경우 buffer는 완충기머리부에 있는 b_data마당의 복사본일뿐이다. 그러나 요청이 여러 블로크를 포함한 상태이고 이것들의 완충기가 기억기에서 련속하지 않으면 완충기는 그림 5-8처럼 완충기머리부의 b_reqnext마당을 통해 련결된다. 읽기의 경우 저준위장치구동프로그람은 buffer에 큰 기억기령역을 할당하고 요청의 모든 분구를 한번에 읽은 다음 자료를 여러 완충기에 복사하는 방법을 선택할수 있다. 쓰기의 경우 저준위장치구동프로그람은 련속하지 않는 많은 완충기로부터 자료를 buffer가 가리키는 단일기억기령역에 복사한 다음 한번에 모든 자료를 전송할수 있다.

그림 5-8은 블로크 세개를 포함하는 요청서술자를 보여준다. 두 블로크완충기는 RAM에서 련속이지만 세번째 완충기는 떨어져있다. 대응하는 완충기머리부는 블로크장치에서 론리적블로크를 나타낸다. 블로크는 반드시 련속해야 한다. 각 론리적블로크에는 분구가 두개 있다. 요청서술자의 sector마당은 디스크에 있는 첫번째 블로크의 첫번째 분구를 가리키고 각 완충기머리부의 b_reqnext마당은 다음완충기머리부를 가리킨다.

초기화과정에서 각 블로크장치구동프로그람은 일반적으로 자기의 처리할 입출력요청을 위해 고정된 수의 요청서술자를 정적으로 할당한다. blk_init_queue()함수는 여유요청서술자가 들어있는 크기가 같은 목록 두개를 설정한다. 하나는 READ연산용이고 하나는 WRITE연산용이다. RAM이 32MB를 넘으면 이 목록의 크기는 64로 설정되고

32MB이하이면 32로 설정된다. 모든 요청서술자의 초기상태는 RQ_INACTIVE이다.

매우 심한 작업부하로 디스크동작이 많을 경우 고정된 수의 요청서술자가 모자랄수도 있다. 자유서술자(free descriptor)가 부족하면 프로쎄스는 진행중인 자료전송이 끝날 때까지 기다려야 한다.

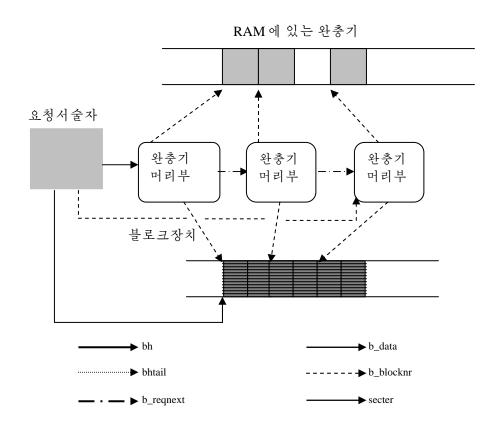



그림 5-8. 요청서술자와 그의 완충기와 분구

따라서 자유 request요소를 기다리는 프로쎄스를 대기시키기 위해 대기렬를 사용한다. get_request_wait()은 자유 요청서술자를 얻으려 시도하고 자유서술자가 없는 경우 현재프로쎄스를 대기렬에 넣어 sleep 상태로 만든다.

get_request()도 비슷하지만 자유요청서술자가 없는 경우 단순히 NULL을 반환한다. 핵심부부하를 줄이기 위해 batch_requests라는 림계값변수(RAM크기에 따라 32 또는 16으로 설정된다.)를 사용한다. 요청서술자를 해제할 때 빈 서술자수가 batch_requests이상 되지 않으면 자유 요청서술자를 기다리는 프로쎄스를 깨우지 않는다. 반대로 자유요청서술자를 찾을 때 get_request_wait()은 자유서술자가 batch_requests보다 적으면 CPU를 풀어준다.

○ 요청대기렬서술자

요청대기렬서술자를 사용하여 요청대기렬를 나타낸다. 각 서술자는 request_queue_t자료구조이고 이 자료구조의 마당은 표 5-18과 같다.

표 5-18. 요청대기렬서술자의 마당

형	마 당	설 명
struct reqrest_list []	rq	요청의 READ, WRITE자유목록
strct list_head	queue_head	대기중인 요청의 목록
elevator_t	elevator	승강기알고리듬함수
request_fn_proc*	request_fn	구동프로그람의 전략루틴
merge_request_fn*	back_merge_fn	블로크를 요청에 덧붙이는 함수
merge_repuest_fn*	front_merge_fn	블로크를 요청의 앞에 삽입하는 함수
merge_requests_fn*	merge_requests_fn	요청을 확대하여 린접한것과 합치는 함수
make_request_fn*	make_repuest_fn	구동프로그람에 요청을 전달하는 함수 (보통 적절한 대기렬에 요청을 삽입한다.)
Plug_device_fn *	Plug_device_fn	구동프로그람 련결(plug)하는 함수
void *	queuedata	장치구동프로그람의 내부자료
struct tq_struct	Plug_tq	련결(plug)기법을 위한 작업대기렬항목
char	plugged	구동프로그람이 련결되였는지 나타내는 기발
char	head_active	구동프로그람이 련결되지 않았을 때 대기렬의 첫번째 요청이 활성화되여있는지 나타내는 기발
spinlock_t	qucue_lock	요청대기렬잠그기
wait_queue_head_t	wait_for_request	요청서술자가 없을 때 기다리는 대기렬

핵심부가 장치구동프로그람을 초기화할 때 구동프로그람이 처리할 각 요청대기렬에 대해 요청대기렬서술자를 생성하고 초기화한다.

요청대기렬은 그 요소가 요청서술자(즉 요청자료구조)인 2중련결목록이다. 각 요청대

기렬서술자의 queue_head마당은 목록의 머리부(첫번째 묶음요소)를 보관하며 요청서술자의 queue마당에 있는 지적자는 요청을 목록에 있는 이전 다음요소와 련결한다. 렬목록에 있는 요소의 순서는 블로크장치구동프로그람에 따라 다르다. Linux핵심부는 두 종류의 정의된 요소순서를 제공한다. 이에 대해서는 《요청대기렬확장》에서 설명한다.

○ 블로크장치 저준위구동프로그람서술자

각 블로크장치구동프로그람은 요청대기렬을 하나 이상 정의할수 있다. 각 구동프로그람의 요청대기렬를 관리하기 위해 저준위구동프로그람서술자를 사용한다. 이 서술자는 blk_dev_struct자료구조이며 마당은 표 5-19와 같다. 모든 블로크장치의 서술자는 blk_dev표에 보관하고 블로크장치의 주번호를 사용하여 참조한다.

표 5-19. 블로크장치 구동프로그람서술자의 마당

형	마 당	설 명
request_queue_t	request_queue	공통요청대기렬(장치별 대기렬을 정의하지 않은 구동프로그람에서 사용)
queue_proc *	queue	장치별 대기렬의 주소를 반환하는 메쏘드
void *	data	queue가 사용하는 자료(례를 들면 부번호)

블로크장치구동프로그람이 모든 물리적블로크장치에 대해 요청대기렬를 하나만 가지고있다면 해당 렬의 주소를 request_queue마당에 보관한다. 블로크장치구동프로그람이 여러 렬을 관리한다면 queue마당이 독자적인 구동프로그람 메쏘드의 주소를 가리킨다. 이 메쏘드는 블로크장치파일의 식별자를 파라메터로 받아서 data마당의 값에 따라 렬을 선택하여 적절한 요청대기렬의 주소를 반환한다.

9. Il_rw_block()함수

ll_rw_block()함수는 블로크장치요청을 생성한다. 이 함수는 블로크 하나이상의 입출력자료전송을 시작하도록(trigger) 핵심부의 여러 곳에서 호출된다. 이 함수는 다음과 같은 파라메터를 받는다.

연산류형인 rw.READ, WRITE, READA중 하나를 선택할수 있다. 마지막연산류형은 다른 류형과 달리 사용가능한 요청서술자가 없을 때 차단되지 않고 즉시 완료한다.

전송할 블로크수인 nr블로크(모두 블로크크기가 같고 같은 블로크장치를 나타내야 한다.)를 나타내는 완충기머리부를 가리키는 지적자 nr개를 포함하는 배렬인 bhs완충기머리부는 각각이 블로크번호, 블로크크기, 가상장치식별자 등을 나타내도록 미리 초기화된다.(앞에서 본 《완충기머리부》 참고)

함수는 다음과 같은 연산을 수행한다.

1. bhs 배렬의 각 완충기머리부에 대해 블로크크기 b_size가 가상장치 b_dev의

블로크크기와 일치하는지 검사한다.

2. 연산이 WRITE인 경우 블로크장치가 읽기전용이 아닌지 검사한다.

3. bhs 배렬의 각 완충기머리부에 대해 다음과 같은것을 실행한다.

a. 완충기머리부의 BH_Lock기발을 설정한다. 다른 핵심부스레드에 의해 이미 설정되여있으면 이 완충기를 건너뛴다.

b. 완충기머리부의 b_count마당을 증가시킨다.

c. 완충기머리부의 b_end_io마당을 end_buffer_io_sync()즉 자료전송이 완료되였을 때 완충기머리부를 갱신하는 함수로 설정한다.(뒤에 나오는 《저준위요청처리》참고)

d. 블로크에 쓰기를 해야 하면 완충기머리부의 BH_Dirsy기발을 다음중 한가지 방법으로 검사한다.

- BH_Dirty가 설정되여있지 않으면 b_end_io메쏘드(end_buffer_io_sync()함수)를 실행하고 이 블로크를 쓸 필요가 없으므로 다음완충기에 대해 계속한다.

- BH_Dirty가 설정되여있으면 이 기발을 해제하고 이 완충기머리부를 잠그기가 걸려있는 완충기머리부의 목록에로 옮긴다.

일반적으로 ll_rw_block()를 호출하는 쪽에서 쓰려고 하는 각 블로크에 대해 BH_Dirty기발을 설정해야 한다.

따라서 기발이 설정되여있지 않는 경우 ll_rw_block()은 블로크가 이미 쓰기연산중이라고 판단하고 아무일도 하지 않는다.

e. 블로크에서 읽기를 해야 하면 완충기머리부의 BH_Uptodate기발을 검사한다.기발이 설정되여있으면 b_end_io메쏘드(end_buffer_io_sync()함수)를 실행하고 다음완충기에 대해 계속한다. 완충기가 유효한(최신)자료를 담고있으면 핵심부는 디스크에서 블로크를 절대로 다시 읽지 않는다.

f. Submir_bh()함수를 호출한다. 이 함수는 다음과 같이 수행한다.

- 디스크에서 첫번째 블로크의 분구수를 계산한다. 즉 b_blocknr(론리적블로크번호)와 b_size(블로크크기)에서 b_rsector마당값을 계산한다. 이 마당은 블로크장치구동프로그람이 LVM(론리볼륨관리자)이거나 RAID디스크를 처리하면 변경될수 있다.

- b_state의 BH_Req기발을 설정하여 요청중인 블로크임을 표시한다.

- b_dev마당의 b_rdev마당을 초기화한다. 이 마당은 블로크장치구동프로그람이 LVM나RAID디스크를 처리하면 변경될수 있다.

4.generic_make_request()를 호출한다.

generic_make_request()함수는 요청을 저준위구동프로그람에 전달한다. 함수는 완충기머리부 bh와 연산류형 rw(READ, WRITE, READA)를 파라메터로 받아서 다음연산을 수행한다.

a. bh->b_rsector가 블로크장치의 분구수를 넘지 않는지 검사한다. 넘는다면 핵심부오유통보를 출력하고 완충기머리부의 b_end_io메쏘드를 호출하고 완료한다.

b. bh->b_rdev에서 블로크장치구동프로그람의 주번호 maj를 추출한다.

c. 저준위구동프로그람서술자 blk_dev[maj]에서 장치구동프로그람요청대기렬의 서술자를 얻는다.

이를 위해 blk_dev[maj].queue 메쏘드가 정의되여있으면 이것을 호출한다. 그렇지 않으면 blk_dev[maj].request_queue 마당을 읽는다.(이 경우 구동프로그람은 단일렬를 사용한다.)

d. 앞에서 찾은 요청대기렬서술자의 make_request_fn 메쏘드를 호출한다.

대부분의 경우 블로크장치구동프로그람에서 make_request_fn메쏘드는 __make_request()함수호출로 구현한다. 이 함수는 렬서술자완충기머리부 bh연산류형 rw를 파라메터로 받아서 다음연산을 수행한다.

a. 연산의 종류가 READA인지 검사한다. READA이면 rw_ahead기발을 1로 설정하고 rw를 READ로 설정한다.

b. create_bounce()함수를 호출한다. 이 함수는 bh->b_page->flags의 PG_highmem기발값을 읽어 bh->b_data완충기가 웃자리기억기에 있는지 여부를 판단한다. 완충기가 웃자리기억기에 있으면 저준위구동프로그람이 완충기를 처리하지 못할수도 있다. 따라서 create_bounce()가 림시로 아래자리기억기에 새로운 완충기를 할당하여 새로운 완충기머리부가 이 완충기를 가리키도록 한다. 새로운 완충기머리부는 bh와 거의 동일하지만 b_data마당은 새로운 완충기를 가리키고 b_private 마당은 원래완충기머리부 bh를 가리키며 b_end_io메쏘드는 입출력연산이 완료하면 아래자리기억기완충기를 해제하는 독자적인 메쏘드를 가리킨다. rw가 WRITE면 create_bounce()가 아래자리기억기완충기를 웃자리기억기완충기의 내용으로 채우고 READ면 b_end_io메쏘드가 아래자리기억기완충기를 웃자리기억기완충기로 복사한다.

c. 요청대기렬이 비였는지 검사한다.

- 요청대기렬이 비여있다면 새로운 요청서술자를 삽입하고 저준위구동프로그람의 전략루틴을 나중에 활성화하도록 순서짜기한다.

- 요청대기렬이 비여있지 않으면 새로운 요청서술자를 삽입하고 이미 렬에 들어있는 다른 요청과 새로운 요청을 합칠수 있는지 시도해본다. 인차 고찰하겠지만 이 경우 전략루틴의 활성화를 순서짜기할 필요가 없다.

이제 이 두 경우를 더 자세히 보자.

1) 전략루틴의 활성화순서짜기

앞에서 본바와 같이 전략루틴의 활성화를 연기하면 련속되는 블로크에 대한 요청을 하나로 묶을수 있는 가능성을 높이는데 도움이 된다. 활성화를 연기하기 위해 장치접속(plugging)과 차단해제(unpluggjng)이라는 기법을 사용한다. 블로크장치구동프로그람이 접속되여있으면 구동프로그람의 렬에 처리할 요청이 있더라도 전략루틴이 활성화되지 않는다.

실제 장치의 요청대기렬이 비여있고 장치가 이직 접속되지 않았다면 __make_request()가 장치접속을 수행한다. 장치접속을 수행하는 plug_device_fn 메쏘드는 보통 generic_plug_device() 함수를 통해 구현한다. 이 함수는 요청대기렬서술자의 plugged마당을 1로 설정하고 plug_tq과제렬요소(요청대기렬서술자에 정적으로 포함되여 있음)를 tq_disk과제렬에 삽입하여 장치의 전략루틴이 나중에 활성화되도록 한다.

이어서 __make_request()함수는 get_request()함수를 호출하여 새로운 요청서술자를 할당한다. 사용할수 있는 요청서술자가 없다면 함수는 rw_ahead기발의 값을 검사한다. 이 기발이 설정되여있으면 함수가 상대적으로 중요하지 않은 미리읽기연산을 수행하고있는것이므로 입출력자료전송을 실행하지 않고 b_end_io메쏘드를 호출한 다음 완료한다. 그렇지 않으면 함수는 get_request_wait()함수를 호출하여 자유요청서술자가 생길 때까지 잠든다.

다음으로 __make_request()는 새로운 요청서술자를 완충기머리부에서 읽은 정보로 초기화한 다음 적절한 실제장치의 요청대기렬에 삽입하고 완료한다.

실제 입출력자료전송은 어떻게 시작하는가? 핵심부는 tq_disk과제렬이 요소를 하나라도 포함하고있는지 주기적으로 검사한다. 이 검사는 kswapd 같은 핵심부스레드에서 이루어지거나 핵심부가 완충기나 요청서술자와 같은 블로크장치구동프로그람과 관련한 자원을 기다려야 할 때 이루어진다. 핵심부는 tq_disk를 검사하는 동안 렬에서 어떤 요소든 제거하고 대응하는 함수를 호출한다.

보통 plug_tq과제렬에 보관된 함수는 generic_unplug_device()함수를 가리킨다. 이 함수는 요청대기렬서술자의 plugged마당을 0으로 하고 request_fn메쏘드를 호출하여 저준위구동프로그람의 전략루틴을 호출한다. 이 과제를 《장치를 차단해제(unplugging) 한다.》고 한다. 결과적으로 구동프로그람의 렬에 포함된 요청이 처리되여 대응하는 입출력자료전송이 이루어진다.

2) 요청대기렬확장

요청대기렬이 비여있지 않다면 핵심부가 렬에 첫번째 요청을 삽입했을 때 구동프로그람이 이미 접속된것이다. 따라서 전략루틴을 다시 활성화하도록 순서짜기할 필요가 없다. 저준위구동프로그람은 이미 접속해제되여있거나 곧 접속해제될것이다.

__make__request( )가 요청대기렬이 비여있지 않음을 알게 되였을 때 저준위구동프로그람은 렬의 요청을 처리하고있을수도 있다. 그럼에도 불구하고 저준위구동프로그람은 일반적으로 처리하기 전에 이미 렬에서 요청을 제거하기때문에 함수는 안전하게 렬을 수정할수 있다. 그렇지만 특수한 경우 즉 요청대기렬의 head_active마당이 설정되여있으면 함수는 렬의 첫번째 요청을 건드리지 않는다. 이 기발은 저준위구동프로그람이 언제나 렬의 첫번째 요청을 처리하고 입출력자료전송이 완료할 때까지 요청을 렬에서 제거하지 않는 방책을 사용하는 경우에 설정된다.

__make_request()함수는 새로운 요소를 렬에 추가하거나 이미 존재하는 요청과 병

합한다. 이미 존재하는 요청과 합치는 경우를 《 블로크클라스터링(block clustering)》이라고 부른다.

블로크클라스터링을 위해서는 다음조건을 모두 만족해야 한다.

- 삽입될 블로크는 요청안에 있는 다른 블로크와 같은 블로크장치에 속해야 하며 이것들과 련속적이여야 한다. 즉 요청의 처음블로크 바로 앞에 오거나 요청의 마지막블로크 바로 뒤에 와야 한다.

- 요청에 포함된 블로크는 삽입될 블로크와 같은 입출력연산류형(READ 또는 WRITE)이여야 한다.

- 확장된 요청은 허용된 최대분구수를 넘지 말아야 한다. 이 값은 max_sectors표에 보관되여있고 블로크장치의 주번호와 부번호로 참조한다. 기본값은 255분구이다.

- 확장된 요청은 허용된 최대토막수를 넘지 말아야 한다.(《요청서술자》참고) 이 값은 보통 128이다.

- 요청의 완료를 기다리는 프로쎄스가 없어야 한다. 즉 요청서술자의 waiting마당이 NULL이여야 한다.

__make_request()함수는 요청한 블로크를 어떻게 렬에 삽입할지 결정하기 위해 전통적으로 《승강기(elevator) 알고리듬》이라는 기법을 사용한다. 승강기알고리듬은 기본적으로 렬안에서 요소의 순서를 정의한다. 보통 저준위구동프로그람이 요청을 처리할때도 이 순서를 따른다.

각 블로크장치구동프로그람이 독자적인 승강기알고리듬을 정의할수도 있지만 대부분의 블로크장치구동프로그람은 다음중 하나를 사용한다.

◦ **ELEVATOR_NOOP알고리듬**

새로운 요청서술자를 렬의 끝에 삽입한다. 따라서 오래된 요청이 새로운 요청보다 우선시한다. 블로크클라스터링은 요청을 확대할수 있지만 요청을 더 최신으로 만들수는 없다.

이 알고리듬은 여러 요청사이의 공정한 처리시간을 제공한다.

◦ **ELEVATOR_LINUS알고리듬**

렬에 있는 요소의 순서는 대응하는 분구의 블로크장치에서의 위치를 따르기 쉽다. 이 알고리듬은 물리적장치에서 탐색(seek)연산의 수와 범위를 최소화하려고 한다. 그렇지만 알고리듬은 반드시 렬의 마지막위치에 있는 요청이 오래동안 처리되지 않고 남아있는것을 방지하기 위해 《로화(ageing)기법》도 사용해야 한다. 이 기법은 대기렬에 오래 있은것일수록 우선순위를 높이는 방법이다. 블로크를 포함할수 있는 요청을 탐색할때 알고리듬은 렬의 바닥부터 시작하며 아주 오래된 요청을 발견하면 즉시 탐색을 중단한다.

승강기알고리듬은 요청대기렬서술자의 elevator마당에 포함된 다음 세개의 메쏘드를 통해 구현한다.

elevator_merge_fn

렬를 탐색하여 블로크클라스터링의 후보가 될만한 요청을 찾는다. 블로크클라스터링이 가능하지 않으면 새로운 요청을 삽입할 위치를 반환한다. 클라스터링이 가능하면 새로운 요청의 블로크를 포함하기 위해 확대해야 하는 기존의 요청을 반환한다.

elvator_merge_cleanup_fn

블로크클라스터링을 성공하면 호출한다. 렬에서 클라스터링으로 확장된 요청 다음에 있는 모든 요청을 로화해야 한다.(ELEVATOR_NOOP 알고리듬에서 이 메쏘드는 아무일도 하지 말아야 한다.)

elevator_merge_req_fn

핵심부가 렬에 있는 두 요청을 합칠 때 호출한다. 새로 확장된 요청의 나이(age)를 할당해야 한다. (ELEVATOR_NOOP 알고리듬에서 이 메쏘드는 아무 일도 하지 않는다.)

__make_request()함수는 존재하는 요청을 다른 요청의 앞이나 뒤에 추가하기 위해 요청대기렬서술자의 back_merge_fn이나 front_merge_fn메쏘드를 사용한다. 블로크클라스터링연산이 성공적으로 끝나면 __make_request()는 요청대기렬서술자의 merge_request_fn 메쏘드를 호출하여 확장된 요청을 렬의 이전 또는 다음요청과 합칠 수 있는지 검사한다.

10. 저준위요청처리

Linux의 블로크장치처리과정의 가장 저준위에 도달하였다. 이 수준을 전략루틴이 구현하며 이 루틴은 렬에 모인 요청을 만족시키기 위해 물리적블로크장치와 호상작용한다. 앞에서 언급한것처럼 일반적으로 새로운 요청이 빈 요청대기렬에 삽입된 다음 전략루틴을 시작한다. 한번 활성화되면 저준위블로크장치구동프로그람은 렬에 있는 모든 요청을 처리하며 렬이 비면 완료한다.

전략루틴은 간단히 다음과 같이 구현할수도 있다. 렬의 각요소에 대해 그 요청을 처리하기 위해 블로크장치구동프로그람과 호상작용하고 자료전송을 완료할 때까지 기다린다. 다음으로 처리한 요청을 렬에서 제거하고 다음요청항목을 처리한다.

그러나 이와 같은 구현은 매우 비효률적이다. DMA를 사용하여 자료를 전송하더라도 전략루틴은 입출력을 완료할 때까지 기다리면서 자기를 보류해야 한다. 따라서 다른 사용자프로쎄스에는 손해가 된다.(전략루틴은 입출력연산을 요청한 프로쎄스에서 반드시 처리할 필요가 없으며 앞으로 임의의 시점에서 실행할수 있다. tq_disk 과제렬이 전략루틴을 활성화하기때문이다.)

따라서 많은 저준위블로크장치구동프로그람은 다음과 같은 기법을 채택한다.

- 전략루틴은 렬에 들어있는 첫번째 요청을 처리하고 자료전송을 완료하면 새치기가 발생하도록 블로크장치조종기를 설정한다. 그리고 전략루틴은 완료한다.

- 블로크장치조종기가 새치기를 발생시키면 새치기처리기는 하반부(bottom half)

를 활성화한다. 하반부조종기는 렬에서 요청을 제거하고 전략루틴을 다시 실행하여 렬의 다음요청을 처리하도록 한다.

기본적으로 저준위블로크장치구동프로그람을 다음과 같이 세분화할수 있다.

- 한 요청의 각 블로크를 개별적으로 처리하는 구동프로그람
- 한 요청의 여러 블로크를 한꺼번에 처리하는 구동프로그람

두번째 류형의 구동프로그람은 첫번째 류형보다 설계와 작성과정이 훨씬 복잡하다. 물리적블로크장치에서는 분구가 련속적이여도 RAM에 있는 완충기가 반드시 련속적일 필요는 없다. 따라서 두번째 류형의 구동프로그람은 DMA자료전송을 위한 림시령역을 할당해야 하고 림시령역과 요청목록의 각 완충기사이에 《기억기-기억기복사》를 수행해야 한다.

앞에 있는 두 류형의 구동프로그람이 처리하는 요청은 모두 린접한 블로크들로 구성되여있으므로 어느 경우에나 훨씬 적은 탐색(seek)명령을 사용함으로써 디스크성능을 향상한다. 그러나 두번째 류형의 구동프로그람이 탐색명령을 더 감소시켜주는것은 아니며 디스크에서 여러 블로크를 한번에 전송하는것이 디스크성능을 높이는데 그다지 효과적이지 않다.

핵심부는 두번째 류형의 구동프로그람에 대해 아무것도 제공하지 않는다. 구동프로그람은 요청대기렬과 완충기머리부목록을 직접 처리해야 한다. 각 물리적블로크장치는 근본적으로 서로 달라서(례를 들어 유연성구동프로그람은 디스크자리길의 블로크를 묶어서 전체 자리길을 입출력연산 한번으로 전송한다.) 요청을 한꺼번에 어떻게 처리할것인가에 대한 일반적인 가정은 거의 무의미하기때문이다.

그러나 핵심부는 첫번째 류형의 저준위블로크장치구동프로그람에 대해서는 제한된 지원을 제공한다. 따라서 이 종류의 구동프로그람을 좀 더 보기로 하자.

전형적인 전략루틴은 다음과 같은 작업을 수행해야 한다.

1. 요청대기렬에서 현재요청을 얻는다. 모든 요청대기렬이 비였다면 루틴을 완료한다.

2. 현재요청이 일관된 정보를 포함하는지 검사한다. 특히 블로크장치의 주번호가 요청서술자의 rq_rdev마당에 들어있는 값과 일치하는지 비교한다. 또 목록의 첫번째 완충기머리부에 잠그기가 걸려있는지 검사한다. (ll_rw_block()가 BH_Lock기발을 1로 설정해야 한다.)

3. 첫번째 블로크의 자료를 전송하도록 블로크장치조종기를 설정한다. 자료전송방향은 요청서술자의 cmd마당에서, 완충기주소는 buffer마당에서, 처음 분구번호와 전송될 분구수는 sector와 current_nr_sectors마당에서 각각 얻을수 있다. current_nr_sectors는 요청내에 첫번째블로크의 분구수를 포함하며 nr_sectors는 요청의 전체 분구수를 포함한다. 또한 DMA자료전송을 마치면 새치기를 발생하도록 블로크장치조종기를 설정한다.

4. ll_rw_block()가 블로크클라스터링한 블로크장치파일을 처리하는 경우 전송될

블로크의 관리를 위해 요청서술자의 sector마당을 증가시키고 nr_sectors마당을 감소시킨다. 블로크장치의 DMA자료전송완료와 관련한 새치기처리기는(직접 또는 하반부를 통해) end_request()함수(또는 블로크장치구동프로그람의 같은 일을 수행하는 독자적인 함수)를 호출해야 한다. 이 함수는 자료전송이 성공하였다면 파라메터로 1을 받으며 실패하였다면 0을 받는다. end_request()는 다음과 같은 연산을 수행한다.

1.오유가 발생하면(파라메터 값이 0인 경우) 블로크에 남은 분구를 건너뛰기 위해 sector와 nr_sectors마당을 갱신한다. 3a 단계에서 완충기의 내용이 최신이 아니라고 표시하게 될것이다.

2.전송된 블로크의 완충기머리부를 요청목록에서 제거한다.

3. 완충기머리부의 b_end_io메쏘드를 호출한다. ll_rw_block()함수가 완충기머리부를 할당할 때 이 마당을 end_buffer_io_sync()함수의 주소로 채운다. 이 함수는 다음과 같은 두 연산을 수행한다.

a. 자료전송성공 또는 실패에 따라 완충기머리부의 BH_Uptodate기발을 1 또는 0으로 설정한다.

b. 완충기머리부의 BH_Lock, BH_Wait_IO, BH_launder기발을 지우고 완충기머리부의 b_wait마당이 가리키는 대기렬에 들어있는 모든 프로쎄스를 깨운다.

b_end_io마당은 다른 함수를 가리킬수도 있다. 례를 들어 create_bounce()함수가 림시로 완충기를 아래자리기억기에 생성하였다면 웃자리기억기의 원래완충기를 갱신하고 b_end_io마당이 원래 완충기머리부의 b_end_io 메쏘드를 호출하는 적절한 함수를 가리킨다.

4. 요청목록에 또 다른 완충기머리부가 있다면 요청서술자의 current_nr_sectors마당을 새로운 블로크의 분구수로 설정한다.

5.buffer 마당을 새로운 완충기주소(새로운 완충기머리부의 b_data마당에서 얻을수 있다.)로 설정한다.

6.요청목록이 비여있다면 모든 블로크를 처리했으므로 다음연산을 수행한다.

a. 요청대기렬에서 요청서술자를 제거한다.

b. 요청이 완료되기를 기다리는 프로쎄스를 모두 깨운다(요청서술자의 waiting 마당).

c. 요청의 rq_status 마당을 RQ_INACTIVE로 설정한다.

d. 요청서술자를 해제된 요청의 목록에 넣는다.

end_request를 수행한 다음 저준위구동프로그람은 요청대기렬이 비였는지 검사한다. 비여있지 않으면 전략루틴을 다시 실행한다. end_request()는 중복된 두 순환을 실행한다. 외부순환은 요청대기렬의 각 요소에 대해 반복하며 내부순환는 각 요청의 완충기머리부목록의 각 요소에 대해 반복한다. 따라서 전략루틴은 요청대기렬의 각 블로크

에 대해 한번씩 실행된다.

11. 블로크와 페지입출력연산

이제부터는 핵심부가 블로크장치구동프로그람을 어떻게 사용하는지 고찰한다. 그리고 핵심부가 디스크입출력자료전송을 활성화하는 여러가지 경우를 고찰한다. 그렇지만 여기서는 블로크장치에 대한 기본적인 두 종류의 입출력자료전송을 보기로 한다.

o **블로크입출력연산**

이 입출력연산은 장치블로크 하나를 전송한다. 따라서 전송할 장치를 RAM완충기 하나에 보관할수 있다. 디스크주소는 장치번호와 블로크번호로 구성된다. 블로크장치의 주번호, 부번호 그리고 론리적블로크번호의 조합으로 나타내는 특수한 디스크블로크와 완충기가 대응한다.

o **페지입출력연산**

이 입출력연산은 페지를 하나를 채우는데 필요한 수의 블로크를 전송한다. (정확한 수는 디스크블로크의 크기와 페지틀크기에 의존한다.) 페지틀의 크기가 블로크크기의 배수면 입출력연산 한번으로 여러 디스크블로크를 전송한다. 각 페지틀은 한 파일에 속한 자료를 담고있다. 이 자료가 련속한 디스크블로크에 보관되여있을 필요가 없으므로 파일의 색인마디와 파일내에서 편위을 사용하여 나타낸다.

블로크입출력연산은 핵심부가 파일체계에서 블로크 하나(례를 들면 색인마디나 초블로크를 담고있는 블로크)를 읽거나 쓸 때 주로 사용된다. 반면에 페지입출력연산은 주로 파일을 읽고 쓸 때(정규파일과 블로크장치파일 모두), 기억기배치를 통해 파일에 접근할 때 교환 등에 사용된다.

두 종류의 입출력연산 모두 블로크장치에 접근하기 위해 같은 함수들을 사용하지만 핵심부는 같은 함수들을 가지고 다른 알고리듬과 완충화기법을 사용한다.

1) 블로크입출력연산

bread()함수는 블로크장치에서 한 블로크를 읽어서 완충기에 보관한다. 이 함수는 장치식별자, 블로크번호, 블로크크기를 파라메터로 받아들여 블로크를 담고있는 완충기의 완충기머리부지적자를 반환한다.

이 함수는 다음과 같은 연산을 수행한다.

1. getblk()함수를 호출하여 완충기캐쉬라는 쏘프트웨어캐쉬에서 블로크를 찾는다. 캐쉬에 블로크가 없으면 getblk()는 블로크를 위한 새로운 완충기를 할당한다.
2. 자료를 포함하고있는 완충기페지에 대해 mark_page_accessed()를 호출한다.
3. 이미 완충기에 유효한 최신자료가 있으면 함수를 마친다.
4. ll_rw_block()를 호출하여 읽기연산을 시작한다.(《ll_rw_block()함수》참고)
5. 자료전송이 끝날 때까지 기다린다. 이를 위해 wait_on_buffer()라는 함수를 호출한다. 이 함수는 현재프로쎄스를 b_wait대기렬에 넣고 완충기가 잠그기에서 풀릴 때

까지 프로쎄스를 보류한다.

6. 완충기가 유효한 자료를 담고있는지 검사한다. 자료가 유효하다면 완충기머리부의 주소를 반환한다. 그렇지 않으면 NULL지적자를 반환한다.

디스크에 어떤 블로크를 직접 쓰는 함수는 없다. 완충기를 불결이라고 표시하면 나중에 해당 완충기의 내용을 청소하면서 디스크에 기록한다. 사실 쓰기연산은 체계성능에 크게 영향을 주지 않으므로 가능하면 언제나 뒤로 연기한다.(《디스크에 불결한 완충기 기록》 참고)

2) 페지입출력연산

블로크장치는 한번에 한 블로크씩 정보를 전송하는 반면에 프로쎄스주소공간(좀 더 정확하게는 프로쎄스에 할당된 기억기령역)은 페지의 모임으로 정의할수 있다. 이와 같은 불일치는 페지입출력연산을 사용하여 어느 정도 감출수 있다. 페지입출력연산은 다음과 같은 경우 실행된다.

- 프로쎄스가 파일에 대해 read() 또는 write()체계호출을 한 경우
- 프로쎄스가 파일을 기억기에 배치하는 페지위치를 읽을 경우 (《기억기배치》 참고)
- 핵심부가 파일기억기배치관련 불결페지를 디스크로 흘리는 경우(2장 2절 《불결기억기배치페지를 디스크로 흘리기》 참고)
- 교환하여 넣기 또는 교환하여 내보내기인 경우 핵심부가 페지를 전체 내용을 디스크에서 적재하거나 디스크에 보관하는 경우

여러 핵심부함수가 페지입출력연산을 활성화할수 있다. 여기서는 교환페지를 읽거나 쓸 때 사용하는 brw_page()함수를 살펴본다.

brw_page()함수는 다음과 같은 파라메터를 받는다.

rw

입출력연산류형(READ, WRITE 또는 READA)

page

페지서술자주소

dev

블로크장치번호(주번호, 부번호)

b

론리적블로크번호배렬

size

블로크크기

페지서술자는 페지입출력연산과 관련한 페지를 가리킨다. 다른 핵심부조종경로가 이 페지에 접근할수 없도록 페지는 brw_page()를 호출하기 전에 이미 잠그기가 걸려있어야(PG_locked기발을 설정) 한다. 페지는 완충기 4096/size개에 나뉘어 보관하였다고 가정한다. 페지의 i번째 완충기는 dev장치의 b[i]블로크에 대응한다.

함수는 다음과 같은 연산을 수행한다.

1. page->buffers마당을 검사해서 NULL이면 create_empty_buffers()를 수행하여 페지에 포함된 모든 완충기를 위해 림시로 완충기머리부를 할당한다.(이와 같은 완충기머리부는 비동기적이다. 이에 관해서는 4장 3절에 있는 《완충기머리부자료구조》에서 설명한다.). page->buffers마당에 페지의 첫번째 완충기의 머리부주소가 보관된다. 각 완충기머리부의 b_this_page마당은 페지안에 있는 다음완충기의 완충기머리부를 가리킨다. 반면에 page->buffers마당이 NULL이 아니면 핵심부는 림시로 완충기머리부를 할당할 필요가 없다.

사실 이 경우 페지는 이미 완충기캐쉬에 포함된 완충기 몇개를 포함하고있다. 그중 일부는 이미 블로크입출력연산에 사용된것이다.(4장 3절의 《완충기페지》 참고)

2. 페지의 각 완충기머리부에 대해 다음과 같은 단계를 수행한다.

a. 완충기머리부의 BH_Lock기발(입출력자료전송을 위한 완충기를 잠그기)과 BH_Mapped기발(완충기가 디스크에 있는 파일을 배치)을 설정한다.

b. b_blocknr마당에 배렬 b에서 대응하는 요소의 값을 보관한다.

c. 비동기적완충기머리부이므로 BH_Async기발을 설정하고 b_end_io마당페 end_buffer_io_async( )의 지적자를 보관한다.(뒤에서 설명한다.)

3.페지의 각 완충기머리부에 대해 submit_bh()를 호출하여 완충기를 요청한다.(《ll_rw_b1ock()함수》 참고)

submit_bh()함수는 접근하는 블로크장치의 장치구동프로그람을 활성화한다. 앞의 《저준위요청처리》에서 설명한것처럼 장치구동프로그람은 실제자료전송을 처리하고 전송한 모든 비동기적완충기머리부의 b_end_io메쏘드를 호출한다. b_end_io마당은 end_buffef_io_async()함수를 가리키며 이 함수는 다음연산을 실행한다.

1. 입출력연산의 결과에 따라 비동기적완충기머리부의 BH_Uptodate기발을 설정한다.

2. BH_Uptodate기발이 설정되여있지 않으면 블로크전송중에 오유가 발생했으므로 페지서술자의 PG_error기발을 설정한다. 함수는 완충기머리부의 b_page마당에서 페지서술자의 주소를 얻는다.

3. page_update_1ock스핀잠그기를 얻는다.

4. 완충기머리부의 BH_Async와 BH_Lock기발을 해제하고 완충기를 기다리는 각 프로쎄스를 깨운다.

5. 페지의 완충기머리부중 아직도 잠그기가 걸린것이 있다면(즉 입출력자료전송이 아직 완료하지 않았다면) page_update_1ock스핀잠그기를 해제하고 되돌기한다.

6. 그렇지 않다면 page_update_1ock스핀잠그기를 해제하고 페지서술자의 PG_error기발을 검사한다. 기발이 해제되였으면 페지의 모든 자료전송이 성공적으로 끝났으므로 함수는 페지서술자의 PG_uptodate 기발을 설정한다.

7. 페지잠그기를 풀고 PG_1ocked 기발을 해제하고 page->wait 대기렬에서 기다

리는 프로쎄스들을 깨운다. 페지입출력연산이 완료한 다음에도 create_empty_buffers()가 할당한 림시로 완충기머리부가 자동으로 해제되지 않는다. 4장 4절에서 보지만 림시로 완충기는 핵심부가 기억기를 회수하려 할 때 해제된다.

12. 문자장치구동프로그람

문자장치는 복잡한 완충화전략이 필요없고 디스크캐쉬도 사용하지 않으므로 다루기 쉬운 편이다. 물론 문자장치마다 요구사항이 다르다. 하트웨어장치를 구동하기 위해 복잡한 통신규약을 구현해야 하는 경우도 있고 간단히 하드웨어장치의 입출력포구에서 값 몇개만 읽으면 되기도 한다. 례를 들어 다중표구 직렬카드장치(여러 직렬포구를 제공하는 하드웨어장치)의 장치구동프로그람은 모선마우스의 장치구동프로그람보다 훨씬 복잡하다.

그렇지만 같은 주번호를 서로 다른 장치구동프로그람에 할당할수 있으므로 약간 복잡한 문제가 발생한다. 례를 들어 주번호 10은 실시간박자나 PS/2마우스와 같이 여러 장치구동프로그람에서 사용한다.

어떤 문자장치구동프로그람이 현재 사용중인지 파악하기 위해 핵심부는 주번호와 부번호로 참조하는 하쉬표을 사용한다. 하쉬표자료배렬은 cdev_hashtable변수에 보관된다. 여기에는 문자장치서술자 64개의 목록이 보관된다. 각 서술자는 char_device자료구조이고 표 5-20에 이 자료구조의 마당을 주었다.

표 5-20. 문자장치서술자의 마당

형	마　당	설　명
struct list_head	Hash	하쉬표목록의 지시자
atomic_t	Count	문자장치서술자의 사용계수기
dve_t	Dve	문자장치의 주번호, 부번호
atomic_t	Openers	사용되지 않음
struct semaphore	Sem	문자장치를 보호하는 신호기

블로크장치구동프로그람과 미찬가지로 하쉬표가 필요한 리유는 핵심부가 문자장치파일이 이미 열려있는지 검사하는것만으로는 문자장치구동프로그람이 이미 사용중인지 판단할수 없기때문이다. 사실 체계등록부나무에는 경로명은 다르지만 주번호와 부번호가 같은 여러 문자장치파일이 있을수 있다. 그리고 이 파일들은 사실 모두 같은 장치구동프로그람을 가리키는것이다.

문자장치서술자를 가리키는 장치파일을 처음으로 열 때 문자장치서술자가 하쉬표에 삽입된다.

init_special_inode()함수가 이 작업을 수행한다. 이 함수는 어떤 디스크색인마디가 장치파일을 나타낸다는 사실을 판단했을 때 저준위파일체계계층이 호출한다.

init_special_inode()는 하쉬표의 문자장치서술자를 탐색한다. 서술자를 찾을수 없으면 새로운 서술자를 할당하고 이것을 하쉬표에 삽입한다. 함수는 또한 서술자의 주소를 장치파일의 색인마디객체의 i_cdev마당에 보관한다.

《 VFS의 장치파일처리 》 에서 open()체계호출의 봉사루틴에서 호출되는 dentry_open()함수는 문자장치파일의 파일객체에 있는 f_op마당이 def_chr_fops표를 가리키도록 설정한다고 하였다. 이 표는 대부분 비여있으며 chrdev_open()함수가 장치파일의 열기메쏘드를 정의한다. 이 메쏘드는 dentry_open()에서 직접 호출된다.

chrdev_open()함수는 문자장치파일의 주번호에 대응하는 chrdevs표요소에 들어있는 주소를 파일객체의 f_op마당에 기록한다. 그리고 이 함수는 open메쏘드를 다시 호출한다.

주번호가 한 장치구동프로그람에 할당되었다면 메쏘드는 장치구동프로그람을 초기화한다. 그렇지 않고 주번호가 여러 장치구동프로그람사이에 공유되고있으면 메쏘드는 한번 더 장치파일의 부번호로 참조히는 자료구조에서 발견한 주소를 파일객체의 f_op마당에 기록한다.

례를 들어 주번호 10인 장치파일에 대한 file_operations자료구조가 단일련결목록 misc_list에 보관되여있다. 끝으로 장치구동프로그람을 마지막으로 초기화하기 위해 open메쏘드를 호출한다. 한번 열리면 읽기와 쓰기를 위해 문자장치파일에 접근할수 있다. 이를 위해 파일객체의 read와 write메쏘드가 장치구동프로그람의 적절한 함수를 가리킨다. 대부분의 장치구동프로그람은 ioctl파일객체메쏘드를 통해 ioctl()체계호출도 지원한다. 이것을 통해 하드웨어장치에 특수명령을 전송할수 있다.

제 6 장. 망관리

Linux핵심부는 다양한 망기본방식(대표적으로 TCP/IP)을 지원하며 망파케트를 순서짜기하기 위한 다양한 알고리듬을 구현한다. 또한 체계관리자가 경로기, 망문, 방화벽, 웨브봉사기 등을 핵심부준위에서 직접 쉽게 설정할수 있는 프로그람들을 포함하고있다.

현재의 망환경코드는 버클리Unix(BSD)에서 파생되였는데 이것을 Net-4라 부른다.

이 코드는 Linux망환경코드의 네번째 판본이다.

망환경코드는 VFS와 비슷하게 객체를 사용하여 많은 기본방식에 대한 공통대면부를 제공한다. 그러나 VFS와 달리 망환경코드는 층으로 구성되여있으며 매개 층은 린접한 층에 대해 잘 정의된 대면부를 가지고있다. 망을 통해 전송한 자료는 재사용할수 없으므로 캐쉬에 보관하지 않는다. Linux는 효률을 높이기 위하여 자료가 층을 통과할 때 복사하지 않고 원본자료를 보관할 때 매층이 요구하는 조종정보를 보관하는데 충분한 완충기에 보관한다.

한개 장에서 Linux망환경코드를 구체적으로 설명하는 것은 불가능하다. 실제로 전체 핵심부원천의 20%정도가 망환경에 대한 내용으로 구성되여있으므로 Linux망보조체계의 모든 기능, 구성요소와 자료구조체의 이름을 설명하는것은 어렵다.

이 장에서 설명하는 내용은 매우 제한된것이다. 이 장에서는 통신규약가운데서 TCP/IP탄창을 기본으로 설명하며 자료련결층과 망층 그리고 전송층만을 설명한다.

그리고 간단히 설명하기 위해 UDP통신규약에 대하여 집중적으로 론의하며 핵심부가 어떻게 단일데타그람을 보내고받는지 간단히 설명한다.

콤퓨터가 망기판을 리용하여 국부망(LAN, Local Area Network)에 련결되여있다고 가정한다.

첫번째 절에서는 Linux망환경에서 리용하는 중요한 자료구조체를 설명하며 두번째 절에서는 단일데타그람을 보내고받는데 필요한 체계호출과 체계루틴을 간단히 설명할것이다. 마지막 두 절에서는 핵심부가 파케트를 보내고받기 위해 망기판와 어떻게 호상작용하는가를 설명한다.

이 장에서는 독자가 이미 망통신규약, 층 그리고 응용프로그람에 관한 지식을 가지고있다고 가정한다.

망보조체계를 위한 프로그람을 작성하는 일이 매우 어려운 작업이다.이 장에서는 통신규약의 구체적인 내용은 설명하지 않기때문이다. 따라서 다른 조작체계에 이미 존재하는 망프로그람의 실례를(오유를 포함해서) 참고해야 한다. 그리고 고속적이고 효률적인 프로그람을 작성해야지 그렇지 않으면 높은 망부하를 견디지 못할것이다.

제 1 절. 망환경자료구조체

이 절에서는 Linux가 망의 저준위층을 어떻게 실현하는가에 대한 일반적인 개요를 설명한다.

1. 망기본방식

망기본방식은 특정한 콤퓨터망이 어떻게 구성되여있는가를 설명한다 . 기본방식은 명백하게 목적이 정의된 《층(layer)》의 집합을 정의한다. 매층에 있는 프로그람은 서로 공유하는 규칙과 규약 (보통 《통신규약(protocol)》이라고 부른다.)에 따라 통신한다.

표 6-1에서 볼수 있는것처럼 Linux는 다양한 망기본방식를 지원한다.

표 6-1. Linux에서 지원하는 중요한 망기본방식

이 름	망기본방식과 통신규약집합
PF_APPLETALK	Appletalk
PF_BLUETOOTH	Bluetooth
PF_BRIDGE	다중통신규약망다리
PF_DECnet	DECnet
PF_INET	IPS의 IPv4 통신규약
PF_INET6	IPS의 IPv6 통신규약
PF_IPX	Novell IPX
PF_LOCAL, PF_UNIX	Unix 도메인소케트(내부통신)
PF_PACKET	IPS의 IPv4/IPv6통신규약저수준접근
PF_X25	X25

IPS(Internet Protocol Suite)는 인터네트의 망기본방식이다. 인터네트는 세계적으로 콤퓨터국부망 수십만대를 련결한 국제적인 망이다. 종종 IPS에서 정의하는 중요한 통신규약 두개의 이름을 사용해서 《TCP/IP망기본방식》이라고 부르기도 한다.

망대면부카드

망대면부카드(NIC:Network Interface Card)는 대응하는 장치파일이 없는 특별한 입출력장치이다. 기본적으로 망카드는 원격체계와 련결되여있는 선으로 자료를 보내고 원격체계에서 보내온 파케트를 핵심부기억기에 받는다.

BSD에서 시작해서 모든 Unix체계는 체계에 설치된 매개의 망카드에 서로 다른 이름을 할당한다. 례를 들어 첫번째 국부망카드에는 eth0이라는 이름을 할당한다.

그러나 이러한 이름은 어떠한 장치파일과도 대응하지 않으며 체계등록부나무구조에

색인마디를 보관하지 않는다.

파일체계를 리용하지 않으므로 체계관리자는 장치이름과 망주소간의 관계를 설정해야 한다. 《망환경관련체계호출》절에서 볼수 있는것처럼 BCDUnix는 체계호출의 새로운 그룹을 도입했다. 이러한 그룹은 망장치에 대한 표준프로그람모형이 되였다.

2. BSD소케트

일반적으로 모든 조작체계는 사용자방식프로그람과 망환경코드간의 응용프로그람대면부(API: Application Programming Interface)를 정의해야 한다.

Linux망환경API는 BSD소케트를 기반으로 하고있다. BSD소케트는 버클리Unix 4.1c BSD에서 도입했으며 대부분의 Unix계렬에서 제공한다.(직접 실현하거나 사용자방식서고의 형태로 실현한다.)

소케트는 통신의 종단점(endpoint) 즉 두 프로쎄스를 련결하는 통로의 끝에 있는 입출구에 해당된다. 자료를 한쪽 입출구에 넣으면 잠시후 다른 쪽 입출구에 나타난다.

통신하는 두 프로쎄스는 서로 다른 콤퓨터에 위치할수 있으며 두 종단점사이에서 자료를 이동하는것은 핵심부의 망환경코드이다.

Linux는 sockfs특수파일체계에 속하는 파일형태로 BSD소케트를 실현한다.(12장에 있는 《특수파일체계》절 참고) 구체적으로 말하면 핵심부는 새로운 BSD소케트에 대해 sockfs특수파일체계에 새로운 색인마디를 생성한다. BSD소케트의 속성은 socket 자료구조체에 보관된다. 이 자료구조체는 sockfs색인마디의 u.socket_i마당에 포함된 객체이다.

BSD소케트객체의 주요마당은 다음과 같다.

- inode

  sockfs의 색인마디객체를 가리킨다.

- file

  sockfs의 파일의 파일객체를 가리킨다.

- state

  소케트의 련결상태를 보관한다. 소케트련결상태에는 SS_FREE(할당되지 않음), SS_UNCONNECTED(련결되지 않음), SS_CONNECTING(련결중), SS_CONNECTED(련결됨), SS_DISCONNECTING(련결중지중)이 있다.

- ops

  socket객체의 메쏘드를 보관하고있는 proto_ops자료구조체를 가리킨다. 소케트의 메쏘드는 표 6-2와 같다. 대부분의 메쏘드는 소케트에 대해 동작하는 체계호출을 가리킨다.

  매 망기본방식은 자체함수를 리용하여 메쏘드를 구현한다. 그러므로 동일한 체계호출이라도 대상소케트가 속한 망기본방식에 따라 다르게 동작한다.

표 6-2. BSD소케트객체의 메쏘드

메 쏘 드	설 명
Release	소케트를 닫기
Bind	지역주소(이름)할당
Connect	련결을 성립(TCP)하거나 원격주소를 할당(UDP)
socketpair	쌍방향소케트흐름을 위한 소케트쌍생성
Accept	련결요구에 대한 대기
Getname	지역주소얻기
Ioctl	ioctl()함수
Listen	련결요구를 성립하기 위해 소케트를 초기화
Shutdown	전이중련결의 한쪽 또는 량쪽 모두 닫기
setsockopt	소케트기발에 대한 값 설명
getsockopt	소케트기발에 대한 값을 얻기
Sendmsg	소케트를 리용하여 파케트전송
Recvmsg	소케트로부터 파케트를 받기
Mmap	파일기억기배치(망소케트에서는 리용하지 않음)
Sendpage	파일에서(로) 직접 자료를 복사(sendfile()체계호출)

- sk

저준위struct sock소케트서술자를 가리킨다. (다음절 참고)

3. INET소케트

INET소케트는 struct sock형자료구조체이다. ISP망기본방식에 속하는 BSD소케트는 socket객체의 sk마당에 INET소케트의 주소를 보관한다.

socket객체(BSD소케트를 나타냄)가 모든 망기본방식에서 공통적인 마당을 포함하고있기때문에 INET소케트가 필수적이다. 핵심부는 특정한 망기본방식의 소케트에 대해 다른 몇가지 정보를 보관해야 한다. 례를 들어 INET소케트의 경우 핵심부는 지역IP주소와 원격IP주소, 지역/원격포구번호, 전송통신규약, 소케트에서 받은 파케트의 대기렬, 소케트로 보내기 위해 대기중인 파케트의 대기렬, 소케트를 통해 전송하는 파케트를 처리하기 위한 메쏘드의 표 등을 관리해야 하다. 이런 속성들을 INET소케트에 보관한다.

INET소케트객체는 리용할 전송통신규약(TCP 또는 UDP)의 류형에 따라 몇개의 메쏘드를 정의하고있다. 이러한 메쏘드는 proto류형의 자료구조체에 보관되며 표 6-3과 같다.

표 6-3. INET소케트객체의 메쏘드

메 쏘 드	설 명
Close	소케트를 닫기
Connect	련결을 성립(TCP)하거나 원격 주소할당(UDP)
Disconnect	성립된 련결을 끊기
Accept	련결요구를 대기
Ioctl	ioctl()의 명령함수
Init	INET소케트객체생성자
Destroy	INET소케트객체해제자
Shutdown	전이중련결의 한쪽 또는 량쪽을 닫기
Setsockopt	소케트기발에 대한 값설정
Getsockopt	소케트기발에 대한 값을 얻기
Sendmsg	소케트로 파케트전송
Recvmsg	소케트에서 파케트받기
Bind	지역주소(이름)할당
backlon_rev	파케트를 받을 때 호출하는 재귀호출(callback)함수
Hash	Per_protocol하쉬표에서 INET소케트를 추가
Unhash	Per_protocol하쉬표에서 INET소케트를 제거
get_port	INET소케트에 포구번호할당

이 표에서 보는것처럼 많은 메쏘드가 BSD소케트객체의 메쏘드와 동일하다.(표 6-2 참고) 실제로 BSD소케트메쏘드는 대응하는 INET소케트메쏘드가 정의되여있다면 INET소케트메쏘드를 호출한다.

sock객체는 80개이상의 마당을 포함한다. 대부분의 마당은 다른 객체, 메쏘드표나 마당자체의 구체적인 설명을 담고있는 다른 자료구조체를 가리킨다.

4. 목적지캐쉬

《connect()체계호출》에서 설명하겠지만 프로쎄스는 일반적으로 소케트에 《이름을 할당》한다. 다시 말하면 소케트에 기록한 자료를 받을 주콤퓨터의 원격IP주소와 포구번호를 지적한다. 핵심부는 또한 원격주콤퓨터에서 전송된 적당한 포구번호를 가진 파케트를 소케트를 읽는 프로쎄스에 전달한다.

실제로 핵심부는 사용중인 소케트에서 지정한 원격주콤퓨터에 관한 자료를 기억기에 계속 유지하고있어야 한다. 망환경코드의 속도향상을 위해 이러한 자료를 목적지캐쉬라는곳에 보관한다. 목적지캐쉬의 입구점은 dst_entry형 객체이다. 각각의 INET소케트

는 dst_cache마당에 소케트의 목적지주콤퓨터를 나타내는 dst_entry객체를 가리키는 지적자를 보관한다.

dst_entry객체는 핵심부가 해당 원격주콤퓨터에 파케트를 전송할 때마다 사용하는 많은 자료를 보관한다. 례를 들면 다음과 같다.

- 파케트를 전송하거나 받는 망장치(례를 들면 망기판)를 설명하는 net_device객체를 가리키는 지적자
- 파케트를 최종목적지로 전달하기 위한 린접하는 경로기에 관련된 neighbour구조체를 가리키는 지적자(《이웃캐쉬》참고)
- 전송할 모든 파케트에 첨부할 공통머리부를 나타내는 hh_cache구조체의 지적자(《이웃캐쉬》참고)
- 원격주콤퓨터에서 전송한 파케트를 받을 때마다 호출되는 함수의 지적자
- 파케트를 전송할 때마다 호출되는 함수의 지적자

5. 경로배정자료구조체

IP층의 가장 중요한 기능은 주콤퓨터에서 구성된 파케트나 망카드를 통해 받은 파케트를 최종목적지로 전달하는것이다. 경로배정알고리듬은 높은 망부하를 감당할수 있을 정도로 충분히 빨라야 하므로 이 작업은 아주 중요하다.

IP경로배정기구는 아주 간단하다. IP주소를 나타내는 32bit정수는 주콤퓨터가 포함된 전체 망을 나타내는 《망주소(network address)》와 망내부에서 주콤퓨터를 나타내는 《주콤퓨터식별자(host identifier)》를 포함한다. IP주소를 적확히 해석하기 위해서 핵심부는 해당 IP주소에 대한 《망마스크(network mask)》를 알아야 한다.

즉 IP주소의 어떤 비트가 망주소를 담고있는지 알아야 한다. 례를 들어 IP주소가 192.160.80.110이며 망마스크가 255.255.255.0이라면 192.160.80.0이 망주소이고 110이 망에서 주콤퓨터를 식별한다. 망주소는 대부분 IP주소의 상위비트에 보관하므로 망마스크는 1로 설정된 비트의 수로 나타낼수도 있다.(이 례에서는 24이다.)

IP경로배정의 주요속성으로 인터네트내부의 모든 주콤퓨터는 자기의 지역망(LAN)내부에 있으면서 목적지망으로 파케트를 전달(forwarding)하는 콤퓨터(경로기라고 부른다.)의 주소만 알면 된다.

례를 들어 netstat -m체계명령이 보여주는 다음경로배정표를 살펴보자.

```
Destination      Gateway    Genmask          Flags MSS    Windowirtt   Iface
192.160.80.0     0.0.0.0      255.255.255.0      U        40         0
        0        eth1
192.160.0.0      0.0.0.0      255.255.0.0        U        40         0
       0         eth0
```

192.50.0.0	192.160.11.1	255.255.0.0	UG	40	0
0	eth0				
0.0.0.0	192.160.1.1	0.0.0.0	UG	40	0
0	eth0				

이 콤퓨터는 두개의 망에 련결되여있다. 하나는 IP 주소가 192.160.80.0이며 망마스크(netmask)가 24bit이며 eth1로 할당되여있는 망대면부카드(NIC)를 리용하고있다. 다른 망은 IP주소가 192.160.0.0 이며 망마스크가 16bit이고 eth0에 할당된 NIC을 리용하고있다.

망 192.160.80.0에 속하며 IP가 192.160.80.110 인 주콤퓨터에 파케트를 보낸다고 가정하자. 핵심부는 경로배정표에서 상위입구점(망마스크에 1의 수가 더 많은 항목)부터 검사한다.

매 입구점에 대해서 목적지(destination)주콤퓨터의 IP주소와 망마스크의 론리곱하기(AND)를 수행한다. 결과가 망목적지IP와 동일한다면 핵심부는 파케트경로배정을 위해 해당 입구점을 리용한다. 우의 경우에서는 처음입구점이 일치하며 파케트는 eth1망장치로 전송된다.

이 경우 고정경로배정표항목의 《관문(gateway)》은 null(0.0.0.0)이다. 이것은 주소가 전송주콤퓨터와 같은 망에 있다는 의미로서 콤퓨터는 망에 있는 주콤퓨터에 직접 파케트를 전송한다. 즉 목적지주콤퓨터의 국부망주소와 함께 파케트를 프레임에 매몰한다.

이 프레임은 망에 있는 모든 주콤퓨터에 방송(broadcast)되지만, NIC는 자신과 다른 국부망주소를 담은 프레임을 자동적으로 무시한다.

이제 파케트를 IP주소가 209.204.146.22 인 주콤퓨터로 보낸다고 가정하자. 이 주소는 원격망(콤퓨터와 직접 련결되여있지 않다.)에 속한다. 표의 마지막 입구점은 모든 주소와 일치한다. 망마스크 0.0.0.0과 론리적인 AND연산을 하면 망주소로 항상 0.0.0.0을 산출하기때문이다. 그러므로 우에 있는 항목으로 해석할수 없는 IP주소는 모두 eth0망장치를 통해서 IP주소가 192.160.1.1 인 기본경로기로 전송된다. 이 경로기는 파케트를 대상목적지에 전달하는 방법을 알고있어야 한다. 파케트는 기본경로기의 국부망주소와 함께 프레임에 매몰된다.

6. 전달정보보관소

전달정보보관소(FIB: Forwarding Information Base) 또는 고정경로배정표(static routing table)는 핵심부가 파케트를 최종목적지에 어떻게 보낼것인가를 결정하는데 참조하는 핵심정보이다. 파케트의 목적지망이 FIB에 포함되여있지 않다면 핵심부는 이 파케트를 전송할수 없다. 그러나 앞에서 본것처럼 FIB는 다른 입구점으로 해석할수 없는 IP주소를 처리하기 위한 기본입구점을 담고있다.

FIB를 실현하는 핵심부자료구조체는 매우 복잡하다. 실제로 경로기는 수백행의 내용을 FIB에 포함하며 대부분이 같은 망장치나 같은 관문을 참조할수도 있다. 그림 6-1은 표가 앞에서 보여준 경로배정표의 4개 입구점을 포함하고있을 경우 FIB의 자료구조를 간단히 표시한것이다. /proc/net/route파일을 읽으면 FIB자료구조에 포함된 자료를 저준위에서 확인할수 있다.

main_table대역변수는 IPS기본방식의 고정경로배정표를 나타내는 fib_table객체를 가리킨다. 2차경로배정표를 정의할수 있지만 main_table이 참조하는 표가 가장 중요하다. fib_table객체는 FIB에대한 연산을 수행하는 몇가지 메쏘드의 주소를 포함하며 fn_hash자료구조를 가리키는 지적자를 보관한다.

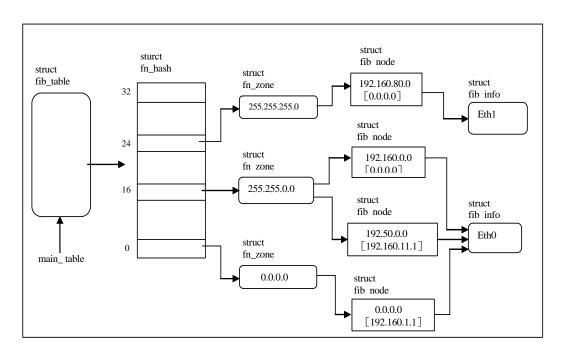



그림 6-1. FIB의 주요자료구조

fn_hash자료구조체는 개개의 FIB구역(zone)을 가리키는 지적자 33개의 배렬이다. 구역에는 주어진 수의 비트가 망마스크에 1로 설정된 목적지망에 대한 경로배정표정보가 있다. 례를 들어 구역24는 마스크 255.255.255.0 을 가진 망입구점을 포함하고있다.

매 구역은 fn_zone서술자로 표현된다. 이것은 하쉬표를 통해 주어진 망마스크를 가진 경로배정표의 입구점에 대한 모임을 나타낸다. 례를 들어 그림 6-1에서 기본구역 16은 192.160.0.0과 192.50.0.0 입구점을 나타낸다.

매개의 경로배정표입구점에 관련한 자료는 fib_node서술자에 보관된다. 경로기는 여러 입구점을 포함할수 있지만 매우 적은 수의 망장치를 가지고있다.

그러므로 공간랑비를 피하기 위해 fib_node서술자는 망카드에 대한 정보를 포함하

지 않으며 여러 입구점이 공유하는 fib_info서술자를 가리키는 지적자를 포함한다.

7. 경로배정캐쉬

고정경로배정표에서 경로를 찾는것은 매우 느린 작업이다. 핵심부는 FIB에 있는 여러 구역을 검색해야 하며 구역의 매 입구점에 대해 주콤퓨터목적지주소와 입구점의 망마스크를 론리곱하기한 결과가 입구점의 망주소와 일치하는가를 확인해야 한다.

핵심부는 경로배정속도를 높이기 위하여 가장 최근에 발견한 경로들을 경로배정캐쉬에 보관한다.

일반적으로 캐쉬는 입구점을 수백개 포함하고있으며 자주 리용하는 경로를 보다 빠르게 검색할수 있도록 정렬되여있다. /proc/net/rt_cache파일에서 캐쉬에 보관된 입구점을 볼수 있다.

경로배정캐쉬의 주요자료구조는 rt_hash_table하쉬표이다. 하쉬함수는 목적지주콤퓨터의 주소와 파케트의 원천주소, 요구하는 봉사형태 등을 조합한다. Linux망환경코드는 경로배정프로쎄스를 세밀하게 조정할수 있도록 한다. 례를 들어 파케트가 어디에서 왔는가, 어떤 종류의 자료를 전송하고있는가에 따라 여러 경로를 통해 전송할수 있게 한다.

캐쉬의 매 입구점은 rtable자료구조체를 가진다. 이 자료구조체는 여러 정보를 보관하며 다음과 같은 정보를 포함한다.

- 원천과 목적지IP주소
- 망문IP주소(있는 경우)
- 입구점으로 구분되는 경로와 관련된 자료

rtable자료구조체의 dst_entry에 보관된다.(《목적지캐쉬》참고)

8. 이웃캐쉬

망환경코드의 다른 핵심요소는 《이웃캐쉬(neighbor cache)》이다. 이 캐쉬는 콤퓨터에 직접 련결된 망에 속하는 주콤퓨터와 관련한 정보를 포함한다.

앞에서 본것처럼 IP주소는 망층의 주요주콤퓨터식별자이다. 그렇지만 IP주소는 저준위자료련결층에는 아무런 의미도 없으며 자료련결층의 통신규약은 하드웨어에 따라 다르다. 핵심부가 어떤 주어진 망카드장치를 리용하여 파케트를 전달해야 하면 하드웨어에 따르는 원천과 목적지망카드장치의 식별자를 포함하는 프레임에 전달할 자료를 매몰해야 한다.

대부분의 LAN은 IEEE801통신규약에 기초하고있으며 특히 일반적으로 《국부망》으로 알려진 802.3통신규약을 따른다. 802통신규약의 망식별자는 48bit의 수이며 반점으로 구분되여있는 6B로 표현한다.(《00:50:DA:61:A7:83》과 같다.) 식별자가 동일한 망카드는 없다.(물론 같은 LAN에 있는 망카드의 식별자들이 서로 다르기만 하면 충분하다.)

핵심부는 주소해석통신규약(ARP: Address Resolution Protocol)이라는 IPS통신규약을 리용한다.

핵심부는 국부망에 다음과 같은 질문을 포함한 방송파케트를 전송한다. 《IP주소 X에 대응하는 망장치의 식별자는 무엇인가?》 그 결과로 이 IP주소를 사용하고있는 주콤퓨터는 망카드식별자를 파케트에 실어 응답한다.

전송할 모든 파케트에 대해 이 작업을 반복하는것은 시간랑비이다. 그러므로 핵심부는 망카드장치식별자와 원격장치에 대한 물리적련결과 관련된 중요한 정보를 이웃캐쉬(arp캐쉬라고도 부른다.)에 보관한다. 이 캐쉬의 내용은 /proc/net/arp파일을 보면 알수 있다. 체계관리자는 arp명령을 사용하여 이 캐쉬의 입구점을 정확히 설정할수 있다.

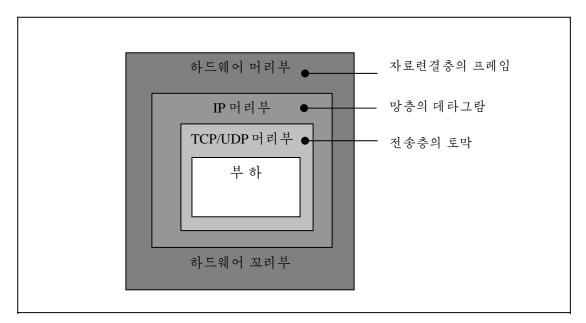
이웃캐쉬의 각 입구점은 neighbour형객체이다. 매 입구점에서 가장 중요한 마당은 ha로서 망카드장치식별자가 보관되여있다. 입구점에는 또한 하드웨어머리부캐쉬에 속하는 hh_cache객체를 가리키는 지적자가 보관된다. 동일한 원격망카드장치에 전송되는 모든 파케트는 동일한 머리부(기본적으로 전송하는 장치식별자와 목적지장치식별자를 전송함)를 가진 프레임에 매몰되므로 핵심부는 모든 파케트에 대해 머리부를 재생성하지 않도록 완충기억기에 머리부를 복사해둔다.

9. 소케트완충기

망장치를 통해 전송되는 단일파케트는 여러 토막의 정보로 구성되여있다.

부하(payload)외에도 자료련결층에서 전송층까지의 모든 망층은 몇개의 조종정보를 추가한다. 망카드장치가 처리하는 파케트의 형태는 그림 6-2와 같다.

그림 6-2. 파케트형태

교육성 교육정보쎈터

전체 파케트는 여러 단계에서 여러 함수를 통해 만들어진다. 례를 들어 UDP/TCP 머리부와 IP머리부는 각각 IPS기본방식의 전송층과 망층에 속하는 함수가 생성한다. 반면에 IP데타그람을 내포하는 프레임을 생성하는 하드웨어머리부와 꼬리부는 망카드장치에 따라 적당한 메쏘드가 생성한다.

Linux망환경코드는 《소케트완충기(Socket Buffer)》라는 큰 기억기구역에 매 파케트를 유지한다. 매개의 소케트완충기에는 서술자가 대응한다. 서술자는 sk_buff형자료구조체로서 다음과 같은 자료구조체를 가리키는 지적자를 담고있다.

- 소케트완충기
- 부하 즉 사용자자료(소케트완충기내부)
- 자료련결꼬리부(소케트완충기내부)
- INET소케트(sock객체)
- 망장치의 net_device객체
- 전송층머리부의 서술자
- 망층머리부의 서술자
- 자료련결층머리부의 서술자
- 목적지캐쉬 입구점(dst_entry객체)

sk_buff자료구조체는 파케트전송에 사용하는 망통신규약의 식별자, 검사합마당, 받은 파케트의 도달시간과 같은 다른 마당들도 포함한다.

일반적으로 핵심부는 자료복사를 하지 않으며 간단히 sk_buff서술자지적자(즉 소케트완충기를 나타낸다.)를 매 망환경층에 차례로 전달한다. 례를 들어 파케트를 보내려고 준비하는 과정에서 전송층은 부하를 사용자방식완충기에서 소케트완충기의 뒤부분에 복사한다. 전송층은 부하앞에 TCP나 UDP머리부를 추가한다. 다음 망층으로 넘어가서 소케트완충기서술자를 전달받아 IP머리부를 전송머리부앞에 추가한다.

끝으로 자료련결층이 머리부와 꼬리부를 추가하고 파케트를 전송하기 위해 대기렬에 넣는다.

제 2 절. 망관련체계호출

망환경과 관련있는 모든 체계호출을 론의할수는 없다. 여기서는 UDP데타그람을 전송하는데 필요한 기본적인 체계호출에 관해서만 설명한다.

대부분의 Unix계렬체계에서 데타그람을 전송하는 사용자방식코드부분은 다음과 같다.

```
I  nt sockfd; /*소케트서술자*/
struct sockaddr_in addr_local, addr_remote; /*IPv4 주소서술자*/
const char *mesg[]= ” Hello, how are you?” ;
```

```
sockfd = socket(PF_INET, SOCK_DGRAM, 0);

addr_local.sin_family = AF_INET;
addr_local.sin_port = htons(50000);
addr_local.sin_addr.s_addr = htonl(0xc0a050f0); /*192.160.80.240*/
bind(sockfd, (struct sockaddr *) &addr_local,
          sizeof(struct sockaddr_in));

addr_remote.sin_family = AF_INET;
addr_remote.sin_port = htons(49152);
inet_pton(AF_INET, "192.160.80.110", &addr_remote.sin_addr);
connect(sockfd, (struct sockaddr *) &addr_remote,
          sizeof(struct sockaddr_in));
write (sockfd, mesg, strlen (mesg)+1);
```

물론 우의 코드는 완전한 원천코드가 아니다. 다시 말하여 main()함수를 정의하지 않았고 머리부파일을 읽기 위한 적확한 #include지시문(directive)을 생략했으며 체계호출의 반환값을 확인하지 않았다. 그러나 우의 원천코드는 UDP데타그람을 전송하기 위해 프로그람이 호출하는 모든 망관련체계호출을 포함하고있다.

이제 프로그람에서 사용한 체계호출을 사용한 순서대로 설명한다.

1. socket()체계호출

socket()체계호출은 둘 이상 프로쎄스가 서로 통신하기 위한 새로운 종단점(endpoint)을 생성한다. 우의 실례에서는 다음과 같이 호출한다.

sockfd = socket(PF_INET, SOCK_DGRAM, 0);

socket()체계호출은 파일서술자를 되돌린다. 실제로 소케트는 일반적인 read()와 write() 체계호출을 리용하여 자료를 읽고쓸수 있으므로 열린 파일과 비슷하다.

socket()체계호출의 첫번째 파라메터는 통신에 리용할 망기본방식과 망기본방식에 적용된 특정한 망층통신규약을 나타낸다. PF_INET마크로는 IPS기본방식과 IP통신규약(IPv4)을 나타낸다. Linux는 [표 6-1]와 같이 다양한 망기본방식를 지원한다.

두번째 파라메터는 망기본방식에서 정의된 통신에 대한 기본적인 모형을 지정한다. 이미 알고있는것처럼 IPS기본방식은 다음과 같은 통신모형에 대한 두개의 선택적인 모형을 제공하고있다.

SOCK_STREAM

TCP전송통신규약으로 실현한 믿음성있고 련결지향적인 스트림기반통신

SOCK_DGRAM

UDP전송통신규약으로 실현된 믿음성없고 비련결지향적인 데타그람기반통신

그리고 특별한 SOCK_RAW값은 망층통신규약에 직접적으로 접근하는데 사용할수 있는 소케트를 생성한다. (우의 경우에서는 IPv4통신규약이다.)

일반적으로 망기본방식은 통신을 위해 다른 모형을 제공할수 있다. 례를 들어 SOCK_SEQPACKET는 믿음성있고 련결지향적인 데타그람통신을 지정한다. 반면에 SOCK_RDM은 믿음성있고 비련결지향인 데타그람통신을 지정한다. 그러나 이것들은 모두 IPS에서는 리용할수 없다.

socket()체계호출의 세번째 파라메터는 통신에서 사용할 전송통신규약을 지정한다. 일반적으로 망기본방식은 매개의 통신모형에 대해 여러 다른 통신규약을 제공할수 있다.

이 값을 0으로 하면 SOCK_STREAM에는 TCP전송통신규약(IPPROTO_TCP)을 선택하며 SOCK_DGRAM에는 UDP통신규약(IPPROTO_UDP)을 선택한다. SOCK_RAW는 IPS의 망층봉사통신규약중 하나를 프로그람작성자가 지정할수 있게 해준다.

례를 들면 인터네트조종통보문통신규약 (Internet Control Message Protocol, IPPROTO_ICMP), 외부관문통신규약 (Exterior Gateway Protocol, IPPROTO_EGP) 또는 인터네트그룹관리통신규약 (Internet Group Management Protocol, IPPROTO_IGMP) 등이다.

socket()체계호출은 sys_socket()체계루틴을 리용하여 실현한다. 이 봉사루틴은 다음 세가지 작업을 수행한다.

1. 새로운 BSD소케트에 대한 서술자할당(《BSD소케트》 참고)
2. 지정한 망기본방식, 통신모형 그리고 통신규약에 따라 새로운 서술자초기화
3. 프로쎄스에서 첫번째 리용가능한 파일서술자를 할당하고 새로운 파일객체를 파일서술자, 소케트객체와 련결한다.

a. 소케트초기화

socket()체계호출의 봉사루틴을 보면 다음과 같다.

새로운 BSD소케트를 할당한 후에 함수는 지정한 망기본방식, 통신모형 그리고 통신규약에 따라 반드시 초기화를 수행해야 한다.

알려진 모든 망기본방식에 대해 핵심부는 net_families배렬에 net_proto_family형 객체지적자를 보관한다. 이러한 객체는 핵심부가 망기본방식에 대한 새로운 소케트를 초기화할 때마다 호출되는 create메쏘드를 정의하고있을뿐이다.

PF_INET기본방식의 create메쏘드는 inet_create()로 실현된다. 이 함수는 socket()체계호출이 파라메터로 지정한 통신모형과 통신규약이 IPS망기본방식과 호환

가능한가를 확인한다. 그 다음 새로운 INET소케트를 할당하고 초기화하며 부모BSD소케트와 련결한다.

b. 소케트파일

socket()봉사루틴은 완료하기 전에 소케트의 sockfs파일에 대해 새로운 파일객체와 새로운 입구점객체를 할당한다. 그 다음 이 객체들을 새로운 파일서술자를 리용하여 체계호출을 실행한 프로쎄스와 련결한다. (2장의 《프로쎄스관련파일》참고)

VFS와 관련시켜보면 소케트에 대응하는 파일은 전혀 특별하지 않다.

대응하는 입구점객체와 색인마디객체는 각각 입구점캐쉬와 색인마디캐쉬에 포함된다. 소케트를 생성한 프로쎄스는 열린 파일에 대해서 동작하는 체계호출, 즉 파라메터로 파일서술자를 받는 체계호출을 리용하여 파일에 접근할수 있다. 물론 파일객체의 메쏘드는 파일이 아닌 소케트에 대해서 동작하는 함수들로 실현되여있다. 그러나 사용자방식프로쎄스와 관련시켜보면 소케트파일은 약간 독특한 면이 있다.

프로쎄스는 파일에서와 같이 open()체계호출을 절대로 호출하지 않는다. 체계등록부나무구조에 대응하는 내용이 없기때문이다.(sockfs특수파일체계는 리용가능한 마운트지점이 없다.) 같은 리유로 unlink()체계호출을 리용하여 소케트파일을 삭제할수 없는데 그것은 sockfs파일체계에 속하는 색인마디는 소케트가 닫기면(또는 해제되면) 핵심부가 자동으로 해제시키기때문이다.

2. bind()체계호출

socket()체계호출이 완료되면 새로운 소케트가 생성되고 초기화된 상태로 된다.

새로운 소케트는 통신규약, 지역IP주소, 지역포구번호, 원격IP주소 그리고 원격포구번호라는 다섯가지 요소로 식별할수 있는 새로운 통신통로를 나타낸다.

여기까지는 《통신규약》요소만이 설정되여있다. 따라서 사용자방식프로쎄스의 다음 동작은 《지역IP주소》와 《지역포구번호》를 설정하는것이다. 이 두가지 항목은 소케트로 파케트를 보내는 프로쎄스를 식별한다. 그러면 원격체계에 있는 받는 프로쎄스는 누가 통신하고있는가, 응답을 어디로 보내야 하는가를 결정할수 있다.

이 실례프로그람에서 해당하는 내용은 다음과 같다.

```
struct sockaddr_in addr_local;
addr_local.sin_family = AF_INET;
addr_local.sin_port = htons (5000);
addr_local.sin_addr.s_addr = htonl(0xc0a050f0); /*192.160.80.240*/
bind (sockfd, (struct sockaddr *) &addr_local,
            sizeof(struct sockaddr_in));
```

addr_local국부변수는 struct sockaddr_in형이며 소케트에 대해 IPS식별자를 나

타낸다.

이 변수는 다음과 같은 중요한 세 마당을 포함한다.

- sin_family

통신규약계렬(AF_INET, AF_INET6 또는 AF_PACKET 표 6-1과 동일하다.)

- sin_port

포구번호

- sin_addr

망주소

IPS기본방식에서 이 주소는 IP주소를 보관하는 32bit마당인 s_addr로 구성되여있다.

그러므로 이 프로그람에서는 addr_local변수마당은 통신규약상수 AF_INET로 설정하며 포구번호는 50000이고 IP주소는 192.160.80.240이다. 점으로 구분된 IP주소를 16진수로 변환하는 방법은 다음과 같다.

80x86기본방식에서는 수자를 주소가 낮은 바이트가 수자에서 낮은 자리를 나타내도록 표시한다. 반면에 IPS기본방식에서는 주소가 낮은 바이트가 수자에서 높은 자리를 나타내도록 표시한다. 자료를 망바이트순서로 전송하는 것을 보장하기 위해 htons()나 htonl()과 같은 함수를 리용한다. 반면에 ntohs()나 ntohl()과 같은 함수는 수신된 자료를 망바이트순서에서 주콤퓨터바이트순서로 변환하는데 리용한다.

bind()체계호출은 소케트파일서술자와 addr_local의 주소를 파라메터로 받는다.

이 체계호출은 struct sockaddr_in자료구조체의 크기도 받는다. 실제로 bind()는 Unix소케트뿐만아니라 임의의 망기본방식의 소케트를 위해 사용할수 있고 주소의 크기가 다른 소케트류형에서도 리용할수 있다.

sys_bind()봉사루틴은 sock_addr변수의 자료를 핵심부주소공간에 복사하고 파일서술자에 대응하는 BSD소케트객체(struct socket)의 주소를 얻는다. 그리고 bind메쏘드를 호출한다. IPS기본방식에서 이 메쏘드는 inet_bind()함수로 실현된다.

inet_bind()함수는 기본적으로 다음과 같은 연산을 수행한다.

1. bind()체계호출에 전달된 IP주소가 주콤퓨터의 어떤 망카드주소와 일치하는가를 확인하려고 inet_addr_type()함수를 호출한다. 만약 다르면 오유코드를 되돌린다. 그러나 사용자방식프로그람은 특정한 IP주소인 INAPPR_ANY(0.0.0.0)를 넘겨줄수 있으며 이것은 IP전송자의 주소의 할당을 핵심부에 넘기는 방법이다.

2. 만약 bind()체계호출에 넘긴 포구번호가 1,024보다 작다면 사용자방식프로쎄스가 초사용자권한이 있는가를 확인한다.(이것은 CAP_NET_BIND_SERVICE의 특징이다. 20장에서 《프로쎄스자격과 특징》참고) 그러나 사용자방식프로쎄스는 포구번호로 0을 넘겨줄수 있으며 이 경우 핵심부는 임의로 사용하지 않는 포구번호를 할당한다. (아래 참고)

3. INET소케트객체의 rcv_addr과 saddr마당을 체계호출에 전달된 IP주소로 설정

한다.(앞에 있는 마당은 경로배정표를 탐색할 때 사용하며 뒤에 있는 마당은 전송되는 파케트머리부에 포함된다.) 방송이나 다중통로방식과 같은 특정한 전송방식이 아니면 두 마당의 값은 동일하다.

4. INET소케트객체의 get_port통신규약메쏘드를 호출하여 초기화하고 같은 지역에 포구번호와 IP주소를 사용하는 INET소케트가 이미 존재하는가를 검사한다. UDP전송통신규약을 사용하는 IPv4소케트인 경우 이 메쏘드는 udp_v4_get_port()로 실현한다. 이 함수는 탐색속도를 높이기 위해 통신규약마다 하쉬표를 하나씩을 리용한다. 사용자방식프로그람이 포구를 0으로 지정하면 함수는 사용하지 않는 포구번호를 소케트에 할당한다.

5. 지역포구번호를 INET소케트객체의 sport마당에 보관한다.

3. connect()체계호출

사용자방식프로쎄스의 다음연산은 《원격IP주소》와 《원격포구번호》를 설정하여 소케트에 기록된 데타그람을 어디로 보내겠는가를 핵심부가 알수 있게 한다. 이 연산은 connect()체계호출을 실행하여 수행한다.

하지만 사용자방식프로그람이 소케트를 목적지주콤퓨터에 련결(connect)할 필요는 없다. 실제로 프로그람은 sendto()와 sendmsg()체계호출을 리용하여 매번 목적지주콤퓨터IP주소와 포구번호를 지정하여 소케트를 통해 데타그람을 전송할수도 있다. 이와 비슷하게 프로그람은 recvfrom()과 recvmsg()체계호출을 실행하여 UDP소케트에서 데타그람을 받을수도 있다. 그러나 사용자방식프로그람이 read()와 write()체계호출을 리용하여 소케트로 자료를 전송한다면 connect()체계호출이 반드시 필요하다.

실례에서 데타그람을 전송하기 위해 write()체계호출을 사용했으므로 통보문의 목적지를 설정하기 위해 connect()를 호출한다. 이와 관련된 부분은 다음과 같다.

```
struct sockaddr_in addr_remote;
addr_remote.sin_family = AF_INET;
addr_remote.sin_port = htons(49152);
inet_pton(AF_INET, "192.160.80.110", &addr_remote.sin_addr);
connect(sockfd, (struct sockaddr *) &addr_remote,
                sizeof(struct sockaddr_in));
```

프로그람은 addr_remote국부변수에 IP주소 192.160.80.110과 포구번호 49152를 할당하여 초기화한다. 이것은 앞절에서 addr_local변수를 초기화하는것과 비슷하다. 그러나 이번에는 inet_pton()서고함수를 호출하여 점으로 구분된 문자렬형태의 IP주소를 망순서형식의 16진수로 변환한다.

connect()체계호출은 bind()체계호출과 동일한 파라메터를 받는다.

그리고 addr_remote변수의 자료를 핵심부주소공간에 복사하고 파일객체에 대응하

교육성 교육정보쎈터

는 BSD소케트객체(struct socket)의 주소를 검색한다. 그리고 connect메쏘드를 호출한다. IPS기본방식에서는 이 메쏘드를 UDP의 경우 int_dgram_connect()함수, TCP의 경우 inet_stream_connect()함수를 리용하여 실현한다.

앞에서 본 간단한 실례프로그람에서는 UDP통신규약을 리용한다.

따라서 inet_dgram_connect()함수가 어떤 동작을 수행하는가를 설명한다.

1. 소케트에 지역포구번호가 없으면 사용되지 않는 포구번호를 자동으로 할당하려고 inet_autobind()를 호출한다. 이 프로그람의 경우 collect()를 호출하기 전에 bind()체계호출을 실행했다. 하지만 UDP를 리용하는 응용프로그람은 꼭 이렇게 할 필요는 없다.

2. INET소케트객체의 connect메쏘드를 호출한다.

UDP통신규약은 INET소케트의 connect메쏘드를 udp_connect()함수로 호출한다. 이 udp_connect()함수는 다음동작을 수행한다.

1. INET소케트에 이미 목적지주콤퓨터가 설정되여있다면 목적지캐쉬(즉 sock객체의 dst_cache마당이다. 앞에서 본 《목적지캐쉬》 참고)에서 이 목적지주콤퓨터를 삭제한다.

2. ip_route_connect()함수를 호출하여 connect()의 파라메터로 전달한 IP주소가 나타내는 주콤퓨터로 가는 경로를 설정한다. 다음으로 ip_route_output_key()를 호출하여 경로배정캐쉬에서 이 경로에 해당하는 항목을 검색한다.(앞에서 본 《경로배정캐쉬》참고) 만약 경로배정캐쉬에 원하는 항목이 없으면 ip_route_output_key()는 ip_route_output_slow()를 호출하여 FIB에서 적당한 항목을 검색한다. (앞에서 본 《전달보관소》 참고) 이제 이 단계가 끝나서 경로를 찾고 적절한 rtable객체의 주소를 결정했다고 가정하자.

3. INET소케트객체의 daddr마당을 rtable객체에서 찾은 원격IP주소로 초기화한다. 일반적으로 이 주소는 사용자가 connect()체계호출의 파라메터로 지정한 IP주소와 같다.

4. INET소케트객체의 dport마당을 connect()체계호출의 파라메터로 지정한 원격포구번호로 초기화한다.

5. INET소케트객체의 state마당에 TCP_ESTABLISHED값을 넣는다.(UDP에서 사용할 때 이 기발은 INET소케트가 목적지주콤퓨터로 《련결》되였다는것을 나타낸다.)

6. sock객체의 dst_cache입구점을 rtable객체에 들어있는 dst_entry객체의 주소로 설정한다.(앞에서 본 《목적지캐쉬》참고)

제 3 절. 소케트에 파케트쓰기

실례프로그람에서 원격주콤퓨터에 통보문를 보낼 준비를 모두 마쳤다. 통보문전송은 소케트가 자료를 기록하는것으로 간단히 진행된다.

```
write(sockfd, mesg, strlen(mesg)+1);
```

write()체계호출은 sockfd파일서술자에 관련된 파일객체의 write메쏘드를 호출한다. 소케트파일인 경우 이 메쏘드를 sock_write함수를 통해 실현하며 이 함수는 다음 연산을 수행한다.

1. 파일의 색인마디에 들어있는 socket객체의 주소를 결정한다.
2. 《통보문머리부》 즉 여러가지 조종정보를 보관하기 위한 msghdr자료구조체를 할당하고 초기화한다.
3. sock_sendmsg()함수를 호출한다. 파라메터로는 socket객체와 msghdr자료구조체의 주소를 전달한다.

이 함수는 다음동작을 수행한다.

a. scm_send()를 호출하여 통보문머리부의 내용을 검사하고 scm_cookie(소케트조종통보문)자료구조체를 할당하고 여기에 통보문머리부에서 추출한 몇개의 마당을 보관한다.

b. socket객체의 sendmsg메쏘드를 호출한다. 파라메터로는 소케트객체, 통보문머리부, scm_cookie자료구조체의 주소를 전달한다.

c. scm_destroy()를 호출하여 scm_cookie자료구조체를 해제한다.

UDP통신규약을 지정하여 BSD소케트를 설정했기때문에 socket객체메쏘드의 주소는 inet_dgram_ops표에 보관된다. 특히 sendmsg메쏘드는 inet_sendmsg()함수를 호출하는데 이 함수는 BSD소케트에 보관된 INET소케트주소를 뽑아낸 다음 INET소케트의 sendmsg메쏘드를 호출한다.

UDP통신규약을 지정하여 BSD소케트를 설정했기때문에 sock객체의 메쏘드의 주소는 udp_prot표에 보관된다. 특히 sendmsg메쏘드는 udp_sendmsg()함수를 호출한다.

1. 전송층 : udp_sendmsg()함수

udp_sendmsg()함수는 파라메터로 sock객체와 통보문머리부(msghdr자료구조체)의 주소를 받아서 다음동작을 수행한다.

1. udphdr자료구조체를 할당한다. 이 자료구조체는 전송될 파케트의 UDP머리부를 담고있다.
2. 목적지주콤퓨터로 가는 경로를 나타내는 rtable의 주소를 sock객체의 dst_cache마당에서 얻는다.
3. ip_append_data()를 호출한다. 파라메터로 sock객체, UDP머리부, rtable객체

교육성 교육정보쎈터

그리고 전송할 파케트를 생성할 UDP에 고유한 함수의 주소 등 관련된 모든 자료구조체의 주소를 전달한다.

2. 망층 : ip_append_data()함수

IP데타그람을 전송하기 위해 ip_append_data()함수를 사용한다. 이 함수는 다음동작을 수행한다.

1. sock_alloc_send_skb()를 호출하여 새로운 소케트완충기와 여기에 대응하는 소케트완충기서술자를 할당한다.(《소케트완충기》 참고)

2. 소케트완충기안에서 부하(payload)의 위치를 결정한다.(부하는 소케트완충기의 끝부분에 위치하므로 그 위치는 부하의 크기에 따라 달라진다.)

3. UDP머리부를 위한 공간을 비워두고 소케트완충기에 IP머리부를 기록한다.

4. getfrag()를 호출하여 사용자방식완충기에서 UDP데타그람의 자료를 복사한다. udp_getfrag()함수는 필요하다면 자료와 UDP머리부의 검사합을 계산한다.(UDP표준에서는 검사합연산은 선택사항이므로 지정할수 있다.)

5. dst_entry객체의 output메쏘드를 호출한다. 파라메터로는 소케트완충기서술자의 주소를 전달한다.

3. 자료련결층 : 하드웨어머리부를 구성

dst_entry객체의 output메쏘드는 자료련결층의 함수를 호출한다. 이 함수는 파케트의 하드웨어머리부(그리고 필요하다면 꼬리부)를 완충기에 기록한다.

IPS의 dst_entry객체의 output메쏘드는 보통 ip_output()함수를 호출하는데 이 함수는 소케트완충기서술자의 주소 skb를 파라메터로 받는다. 이 함수는 다음동작을 수행한다.

・skb->dst 목적지캐쉬객체의 hh마당을 확인하여 적당한 하드웨어머리부가 캐쉬에 이미 있는가를 확인한다. (《목적지캐쉬》 참고) 마당이 비워있지 않으면 캐쉬는 머리부를 포함하고있으므로 하드웨어머리부를 소케트완충기로 복사한다. 그리고 hh_cache객체의 hh_ouput메쏘드를 호출한다.

・그렇지 않고 skb->dst->hh마당이 비워있으면 머리부를 처음부터 생성해야 한다.

함수는 skb->dst의 neighbour마당이 가리키는 neighbour객체의 output메쏘드를 호출한다. 이 메쏘드는 neigh_resolve_output()함수를 호출한다. 이 함수는 머리부를 구성하기 위해 파케트를 전달할 망카드장치와 관련한 net_device객체의 적당한 메쏘드를 호출한다.

그리고 새로운 하드웨어머리부를 캐쉬에 추가한다.

hh_cache캐쉬의 hh_output메쏘드와 neighbour객체의 output메쏘드는 결국 dev_queue_xmit()함수를 호출한다.

dev_queue_xmit()함수는 이후의 전송을 위해 소케트완충기의 대기렬을 관리한다.

일반적으로 망카드는 느린 장치이고 어떤 순간이든 많은 파케트들이 전송을 기다리고있을수 있다.

Linux핵심부가 고성능경로기에서 활용할수 있는 여러가지 복잡한 파케트순서짜기 알고리듬을 지원하지만 전송을 기다리는 파케트들은 일반적으로 선입선출(FIFO: First In First Out)방책에 따라 처리한다.(따라서 파케트의 대기렬이라는 용어를 사용한다 .) 일반적으로 모든 망카드장치는 전송을 대기중인 파케트의 대기렬을 독자적으로 정의한다. 루프백장치(lo)나 다양한 통로화(tunneling)통신규약을 제공하는 장치와 같은 가상장치인 경우는 례외이다.

소케트완충기의 대기렬은 복잡한 Qdisc객체를 리용하여 실현된다. 이 자료 구조에 의하여 파케트순서짜기함수는 효률적으로 대기렬을 조작할수 있고 전송할 《최상의》파케트를 빠르게 선택할수 있다. 그러나 간단히 설명하기 위해 여기서는 대기렬이 단지 소케트완충기서술자의 목록이라고 하자.

핵심적으로 dev_queue_xmit()는 다음동작을 수행한다.

1. 망장치구동프로그람(이 구동프로그람의 서술자는 소케트완충기서술자의 dev마당에 보관되여있다.)이 전송을 대기중인 파케트의 대기렬을 독자적으로 정의하고있는가를 검사한다.(Qdisc객체의 주소는 net_device객체의 qdisc마당에 보관되여있다.)

2. 소케트완충기를 대기렬에 추가하기 위해 적당한 Qdisc객체의 enqueue메쏘드를 호출한다.

3. qdisc_run()함수를 호출하여 망장치가 대기렬에 있는 파케트를 실제로 전송하도록 한다.

sys_write()체계호출봉사루틴이 실행한 함수의 사슬은 여기서 끝난다. 지금까지 본것처럼 마지막결과는 망카드장치의 전송대기렬에 새로운 파케트를 추가하는것이다.

다음의 절에서는 망카드가 파케트를 어떻게 처리하는가를 설명한다.

제 4 절. 파케트전송과 수신

1. 파케트송신

망카드장치구동프로그람은 핵심부가 새로운 파케트를 전송대기렬에 추가하거나 (앞절에서 설명한 내용) 통신통로에서 파케트를 받았을 때 실행한다. 여기서는 파케트전송을 기본으로 설명한다.

앞에서 본것처럼 핵심부는 망카드장치구동프로그람을 활성화하고 싶을 때마다 qdisc_run()함수를 호출한다. 이 함수는 NET_TX_SOFTIRQ프로그람적인 새치기를

실현하는 net_tx_action()함수를 호출할수도 있다.

기본적으로 qdisc_run()함수는 망카드장치가 현재 작업중인가와 대기렬에 있는 파케트를 전송할수 있는가를 확인한다. 례를 들어 카드가 이미 전송중이거나 파케트를 받고있는중이기때문에 장치가 전송할수 없다면 통신통로가 넘쳐나는것을 방지하기 위해 대기렬을 잠시 멈추거나 NET_TX_SOFTIRQ프로그람적인 새치기를 활성화하고 현재 실행중인 qdisc_run()을 완료한다. 시간이 지난 후에 순서짜기프로그람이 ksoftirqd핵심부스레드를 선택하면 net_tx_action()함수는 qdisc_run()을 호출하여 다시 파케트전송을 시도한다.

특히 qdisc_run()은 다음동작을 수행한다.

1. 파케트대기렬이《정지되였는가》를 확인한다. 즉 net_device망카드객체의 state마당에서 적당한 비트가 설정되여있는가를 확인한다. 만약 정지되였다면 함수가 즉시 완료한다.

2. qdisc_restart()함수를 호출한다.

이 함수는 다음동작을 수행한다.

a. Qdisc파케트대기렬의 dequeue메쏘드를 호출하여 대기렬에서 파케트를 추출한다. 대기렬이 비여있으면 완료된다.

b. 핵심부에서 파케트도청(sniffing)을 실행하고있는가를 검사한다. 실행하고있으면 외부로 나가는 모든 파케트의 복사본을 지역소케트로 전달한다. 이 경우 함수는 dev_queue_xmit_nit()함수를 호출하여 이 작업을 수행한다.

c. 망카드장치를 나타내는 net_device객체의 hard_start_xmit메쏘드를 호출한다.

d. hard_start_xmit메쏘드가 파케트전송에 실패했다면 이 파케트를 다시 대기렬에 추가한다.

3. 대기렬이 비여있거나 hard_start_xmit메쏘드가 파케트전송에 실패했다면 함수는 완료한다. 그렇지 않으면 대기렬에 있는 다음파케트를 처리하기 위해 단계 1로 되돌아간다.

hard_start_xmit메쏘드는 망카드장치에 따라 다르며 파케트를 소케트완충기에서 장치의 기억기로 전송하는 일을 담당한다. 특히 이 메쏘드는 DMA전송의 활성화를 스스로 제한한다. PCI기반망카드에서는 적은 수의 DMA전송을 미리 예약해둔다. 그리고 진행중인 DMA전송이 끝날 때마다 카드가 자동적으로 다른 DMA전송을 활성화한다.

장치의 기억기가 모두 차서 카드에서 파케트를 더는 받아들일수 없다면 메쏘드는 net_device객체의 state마당에서 적당한 비트를 설정하여 파케트대기렬을 멈추도록 한다. 이렇게 함으로써 qdisc_run()함수는 완료하고 나중에 프로그람적인 새치기에 의해 다시 실행된다.

DMA전송이 완료되면 카드는 새치기를 발생시킨다. 대응하는 새치기조종기는 다음동작을 수행한다.

1. 카드가 요구한 새치기를 승인한다.

2. 전송오유를 검사하고 구동프로그람의 통계정보를 갱신하는 등의 작업을 수행한다.

3. 필요하다면 raise_softirq()함수를 호출하여 softirq의 활성화를 순서짜기한다.

4. 대기렬이 멈춰있다면 net_device객체의 state마당에 있는 비트를 지우고 파케트처리를 다시 시작한다.

지금까지 본것처럼 망카드장치구동프로그람은 디스크장치구동프로그람과 비슷하게 동작한다. 실지작업은 새치기조종기와 지연함수에서 처리하며 일반적인 프로쎄스는 파케트전송을 기다리며 차단되지 않는다.

이 장의 대부분은 핵심부가 망파케트의 전송을 처리하는 방법을 중점적으로 설명한다. 이미 망환경코드의 중요한 자료구조체를 간단히 보았다. 그러므로 이제 다른 측면인 망파케트를 받는 방법을 보자.

2. 파케트수신

전송과 수신의 가장 큰 차이점은 파케트가 언제 망카드장치에 도착할것인가를 핵심부가 예상할수 없다는것이다. 그러므로 파케트의 수신을 다루는 망환경코드는 새치기조종기와 지연(deferrable)함수에서 동작한다.

정확한 하드웨어주소(카드식별자)를 가지고있는 파케트가 망장치에 도착했을 때 일어나는 일련의 사건을 보자.

1. 망장치는 장치기억기의 완충기에 파케트를 보관한다. (카드는 일반적으로 원형완충기에 한번에 여러 파케트를 보관할수 있다.)

2. 망장치는 새치기를 발생시킨다.

3. 새치기조종기는 파케트에 대한 새로운 소케트완충기를 할당하고 초기화한다.

4. 새치기조종기는 파케트를 장치기억기에서 소케트완충기로 복사한다.

5. 새치기조종기는 자료련결프레임에 매몰된 파케트의 통신규약을 결정하기 위해 함수(국부망과 IEEE 802.3의 경우 eth_type_trans()함수)를 호출한다.

6. 새치기조종기는 netif_rx()함수를 호출하여 Linux망환경코드에 새로운 파케트가 도달했으므로 처리해달라고 알린다.

물론 새치기조종기는 망카드장치에 따라 다르다. 많은 장치구동프로그람은 체계의 다른 장치에 피해를 주지 않도록 노력하며 소케트완충기를 할당하거나 파케트를 복사하는 등의 시간이 오래 걸리는 작업을 지연함수에서 처리하도록 한다.

netif_rx()함수는 망환경층(앞에서 언급한 망카드장치구동프로그람)의 수신코드에 대한 핵심적함수이다. 핵심부는 망장치에서 수신해서 다양한 통신규약 탄창층에서 처리를 기다리는 파케트들을 위해 CPU별 대기렬을 사용한다. 함수는 일반적으로 새로운 파케트를 이 대기렬에 추가하고 raise_softirq()를 호출하여 NET_RX_SOFTIRQ프로그람적인 새치기의 활성화를 순서짜기한다. (여러 CPU에서 동일한 softirq를 동시에 수

행할수도 있다.때문에 수신한 파케트를 CPU 별 대기렬에 보관한다.)

NEXT_RX_SOFTIRQ프로그람적인 새치기는 net_rx_action()함수로 실현하며 이 함수는 다음동작을 수행한다.

1. 대기렬에서 첫번째 파케트를 추출한다. 대기렬에 비여있으면 완료한다.

2. 자료련결층에서 부호화된 망층통신규약번호를 결정한다.

3. 망층통신규약의 적당한 함수를 호출한다.

IP통신규약의 대응하는 함수는 ip_rcv()다.

이 함수는 다음동작을 수행한다.

1. 파케트의 길이와 검사합을 검사한다. 만약 잘못되였거나 잘리였다면 파케트를 버린다.

2. ip_rcv_finish()를 호출하여 소케트완충기서술자의 목적지캐쉬(dst_entry마당)를 초기화한다. 파케트의 경로를 결정하기 위해 함수는 경로캐쉬에서 경로를 찾아보고 다음으로 FIB에서 경로를 찾아본다.(만약 경로캐쉬에 항목이 없을 경우) 이런 방법으로 핵심부는 파케트를 다른 주콤퓨터로 전달해야 하는가, 전송층의 통신규약으로 전달해야 하는가를 결정할수 있다.

3. 파케트도청이나 다른 입력방책을 적용해야 하는가를 검사한다. 적용해야 한다면 방책에 따라 파케트를 처리한다.

4. 파케트의 dst_entry객체의 input메쏘드를 호출한다.

파케트를 다른 주콤퓨터로 전달해야 한다면 input메쏘드는 ip_forward()함수를 호출하고 그렇지 않으면 ip_local_deliver()함수를 호출한다. 두번째 경우를 보자.

ip_local_deliver()함수는 데타그람이 전달되는 과정에서 단편화되였을수 있기때문에 원래의 IP데타그람을 재조합한다. 다음으로 IP머리부를 읽고 파케트가 속한 전송통신규약류형을 결정한다. 만약 전송통신규약이 TCP라면 함수는 끝으로 tcp_v4_rcv()를 호출하고 전송통신규약이 UDP라면 함수는 끝으로 udp_rcv()를 호출한다.

계속해서 UDP경로를 따라가보자. udp_rcv()함수는 다음동작을 수행한다.

1. udp_v4_lookup()함수를 호출하여 UDP데타그람을 전달할 INET소케트를 찾는다.(UDP머리부안에 있는 포구번호를 찾아본다.) 핵심부는 INET소케트를 하쉬표로 관리하므로 검색연산은 매우 빠르게 수행된다. 만약 UDP데타그람이 관련된 소케트가 없으면 함수는 파케트를 버리고 완료한다.

2. udp_queue_rcv_skb()를 호출한다. 이 함수는 sock_queue_rcv_skb()를 호출하여 파케트를 INET소케트의 대기렬(sock객체의 receive_queue 마당)에 보관하고 sock객체의 data_ready메쏘드를 호출한다.

3. 소케트완충기와 소케트완충기서술자를 해제한다.

INET소케트는 data_ready 메쏘드를 sock_def_readable()함수를 호출하는데 이 함수는 기본적으로 소케트대기렬에 있는 잠든 상태의 프로세스를 깨운다.(sock객체의

sleep마당에 배렬되여있다.)

프로쎄스가 INET소케트를 가진 BSD소케트에서 읽을 때 발생하는 마지막단계는 다음과 같다.

read()체계호출은 소케트의 특수파일과 관련된 파일객체의 read메쏘드를 실행하도록 한다. 이 메쏘드는 sock_read()함수를 호출하며 이 함수는 sock_recvmsg()함수를 호출한다. sock_recvmsg()함수는 앞에서 설명한 sock_sendmsg()와 거의 비슷하다. 이 함수는 BSD소케트의 recvmsg메쏘드를 호출한다. 이 메쏘드(inet_recvmsg()함수)는 INET소케트의 recvmsg메쏘드를 호출한다. 즉 tcp_recvmsg()나 udp_recvmsg()함수이다.

끝으로 udp_recvmsg()함수는 다음동작을 수행한다.

1. skb_recv_datagram()함수를 호출하여 INET소케트의 receive_queue대기렬에서 처음파케트를 추출하고 대응하는 소케트완충기서술자의 주소를 되돌린다. 대기렬이 비여있다면 skb_recv_datagram()함수는(read조작이 차단이 아니라면)현재프로쎄스를 차단한다.

2. UDP데타그람의 검사합이 일치하면 전송도중 통보문이 손상되지 않았는가를 확인한다.(실제로 이 단계는 단계3과 동시에 진행된다.)

3. UDP데타그람의 부하를 사용자방식완충기로 복사한다.

제 7 장. 보안기능

이 장에서는 보안체계의 개념과 필요성을 고찰하고 핵심부에 표준적으로 실현되여있는 대표적인 보안조작체계인 SELinux를 대상으로 하여 보안의 구체적인 실현을 서술한다.

제 1 절. 보안체계의 개념

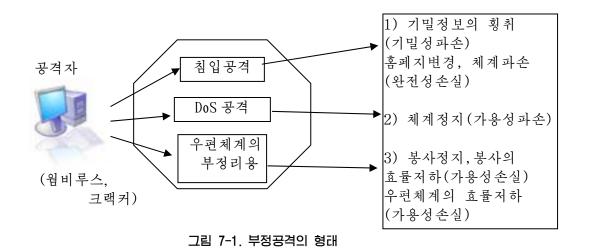
1. 보안조작체계의 일반적인 리해

정보보안관리체계의 국제규격인《ISO/IEC 17799》에 의하면 정보보안은《기밀성》,《완전성》,《가용성》이라고 하는 3대요소를 간단히 묶은것이다.

기밀성 - 권한이 없는 사람에게 정보가 루실되지 않는것. 정보를 도청당한 경우에는 기밀성이 파괴되는것으로 된다.

완전성 - 자료가 정확한 상태로 보존하는것. 홈페지가 변경된 경우에는 완전성이 파괴되는것으로 된다.

가용성 - 허가된 리용자가 항상 필요한 정보를 호출할수 있는것. Web봉사기가 정지하고 리용자가 홈페지 등을 호출할수 없을 때에는 가용성이 파괴되는것으로 된다.

그림 7-1. 부정공격의 형태

부정호출을 진행하는 공격자의 실체는《악의를 가진 인간(크랙커)》과 《웜과 비루스와 같은 부정프로그람》이다. 웜과 비루스는 크랙커가 진행하는 공격을 프로그람에 의해 자동화한것이라고도 생각할수 있다.

이러한 부정호출의 수단을 분류하면 우의 그림에서처럼 《침입공격》, 《DoS공

격》, 《우편체계의 부정리용》의 3종류로 나눌수 있다.

침입공격이라는것은 공격자가 보안구멍을 리용하여 콤퓨터의 조종을 가로채고 홈페지와 체계를 파손하거나 기밀자료를 도청하거나 또는 그 콤퓨터를 리용하여 다른 싸이트에 공격을 진행하는것이다.

DoS(Denial of Service : 봉사불가능)공격이라는것은 대량의 파케트를 보내는 등으로 하여 봉사기가 제공하고있는 봉사를 정지시키는것이다.

우편체계의 부정리용이라는것은 스팸메일을 보내거나 메일을 부정중계하는 공격이다. 이 공격을 받으면 봉사기의 자원이 여분으로 소비되여 버린다.

c. 침입공격

Linux에 대한 침입공격에서는 공격자는 목표로 되는 주콤퓨터의 root권한을 가로챈 후 root권한을 악용하여 파괴활동을 진행한다.

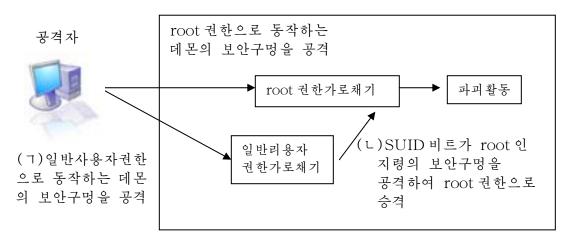



그림 7-2. 침입공격의 동작

우의 그림처럼 root권한 가로채기방법은 크게 두가지이다.

하나는 root권한으로 동작하고있는 데몬프로그람의 보안구멍을 리용하여 그 데몬프로그람을 가지는 방법(직접침입공격)이다. 례하면 sshd와 ftpd 등은 root권한으로 동작하고있다. 이러한 데몬프로그람을 가로채는 방법에는 완충기자리넘침공격이 대표적이지만 그 외에도 각이한 방법이 있다.

또 하나의 방법은 일반리용자권한으로부터 root권한으로《승격》하는 방법(간접침입공격)이다.

그림의 (ㄱ)에서는 일반리용자권한으로 동작하고있는 데몬프로그람의 보안구멍을 리용하여 그 데몬프로그람을 가로챈다. 이 단계에서는 공격자는 아직 일반리용자권한으로

침입할수 있는데 지나지 않는다. (ㄴ)에서 공격자는 SUID비트가 root로 설정되여있는 프로그람을 찾는다. SUID비트가 root로 되여있는 프로그람이라는것은 일반리용자로부터 리용될 때에 일시적으로 root권한으로 동작하는 프로그람인것이다. 례하면 passwd지령을 사용하여 일반리용자는 본래의 root밖에 호출할수 없는 통행암호파일을 바꾸어 쓸수 있다. 이것은 passwd지령이 suid비트가 root로 설정되여있기때문이다. 이러한 SUID비트가 ROOT로 되여있는 프로그람에 보안구멍이 존재하면 공격자는 그 프로그람을 가로채여 일반리용자권한으로부터 root권한으로 승격한다.

침입공격에 대한 현재 주류의 보안대책을 묶으면

- 패치적용과 갱신
- 방화벽의 도입
- 통신내용과 일지의 감시와 경고

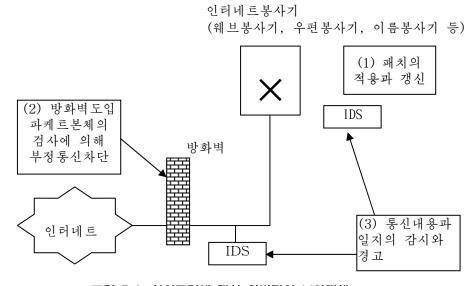



그림 7-3. 침입공격에 대한 일반적인 보안대책

이러한 대책에는 다음의 한계가 있다.

d. **패치적용과 갱신**

프로그람에 부정인자(보안구멍)이 발견될 때마다 그 프로그람에 패치(수정프로그람)을 적용하거나 대책후의 보다 새로운 판본으로 갱신한다. 그러나 패치의 적용은 그리 간단치 않다.

보안구멍이 발견된 후 급속히 패치를 적용할 필요가 있는 경우 기존체계의 동작이 불안정하게 될 위험성도 고려되여야 한다. 이러한것을 평가하는 품은 대단히 크다. 최근에는 보안구멍의 정보공개전에 공격을 받은 경우도 나타난다.

e. 방화벽의 도입

방화벽의 파케트려과기능에 의해 포구번호와 IP주소를 선별기준으로 하여 부정인 통신을 진행하지 않게 한다. 여기서도 파케트려과를 통과하는 통신을 리용한 공격은 방지할수 없다. 례하면 Web봉사기로서 외부망에 공개하고있는 체계에 대하여 포구80번을 리용한 공격을 방지하는것은 곤난하다.

포구번호와 IP주소만이 아니라 통신의 본체를 검사하여 조종을 진행하는 《응용프로그람관문》이라고 하는 형태의 방화벽도 존재한다. 그러나 미리 등록하여놓은 공격밖에 방지할수 없으며 알수 없는 공격에는 대처하기가 곤난하다.

f. 통신내용과 일지의 감시와 경고

구체적으로 침입검사체계(IDS: Intrusion Detection System)을 도입하여 망을 흐르는 파케트와 봉사기의 일지를 감시한다. 공격과 비슷한 패턴의 통신과 일지가 있다면 경보를 발생한다. 그러나 공격패턴을 사전에 등록하여야 하며 등록하고있지 않은 공격을 검출할수 없다. 또한 이 방법에서도 알수 없는 공격은 알아낼수 없다.

여기에 대처할수 있는 기술이 보안OS이다. 보안OS에서는 OS 그 자체를 개조하여 root권한을 무시함으로서 침입공격에 대처한다. 이렇게 되면 침입방법이 아무리 발전하여도 root권한이 존재하지 않기때문에 공격자가 침입하여도 권한을 가로채는 일이 없어진다. 이 방법이라면 기존의 공격만이 아니라 알수 없는 공격에 대해서도 효과적이다.

실지 Linux핵심부에 보안OS를 표준으로 끌어들인다고 하면 아래와 같은 문제가 생긴다.

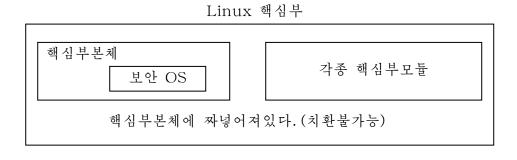



그림 7-4. 핵심부에서 보안체계를 실현하는 방법 1

우의 그림과 같이 Linux핵심부는 핵심부 본체와 모듈로 이루어져있다.

Linux를 보안OS로 하는데는 Linux핵심부본체에 패치를 적용할 필요가 있기때문에 이 대로는 한 종류의 보안OS밖에 Linux핵심부의 표준으로서 끌어넣을수 없다.

그러나 어느 1개만을 선택하는것은 아주 힘들다. 왜냐하면 개개의 보안 OS마다에 특징이 있으며 용도에 따라서 최적의 보안OS가 다르기때문이다. 례하면 높은 보안준위

를 요구하는 경우는 SELinux를 사용하며 거꾸로 사용하기 쉬운것을 우선시하는 경우는 LIDS를 사용한다고 하는것처럼 리용자의 요구에 따라서 유연하게 선택할수 있는것이 편리하다. 따라서 핵심부본체에 짜넣고서는 이것을 실현할수 없다. 거기서 보안OS를 Linux의 핵심부모듈로서 실장하려고 하는 발상이 생기였다.

핵심부모듈로 하면 핵심부본체에는 손을 댈 필요가 없기때문에 리용자는 자기가 좋은 보안OS의 모듈을 선택할수 있게 된다.

핵심부 2.4에서는 보안OS를 모듈로서 실장하는것은 힘들지만 핵심부2.5부터는 <LSM(Linux Scurity Module)>이 핵심부본체에 표준으로서 넣었다.

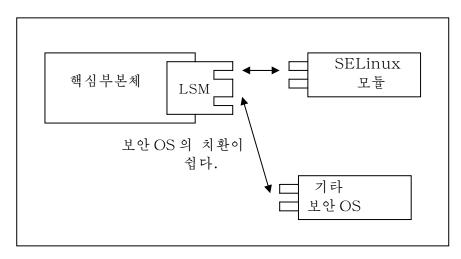



그림 7-5. 핵심부에서 보안체계를 실현하는 방법 2

LSM은 보안OS를 핵심부모듈로서 실장할수 있는 구조를 제공한다. LSM에는 보안체계기능들로서 SELinux, LIDS, DTE, OpenWall 등이 실현되여있다.

2. Linux보안모듈(LSM)

체계보안을 제공하는데서 조작체계보호기구가 노는 중요한 역할을 잘 알고있지만 아직 현재의 조작체계들에 있는 접근조종기구들은 강력한 보안을 충분하게 제공하지 못하고있다. 비록 확장된 많은 접근모형들과 프레임워크들이 제안되고 실현되였어도 주류로 되고있는 조작체계들에서는 대체로 아직 이 강화수법들을 제공하기에 부족점이 많다.

다른 많은 일반목적조작체계들과 같이 2.6이전의 Linux핵심부는 단지 임의의 접근조종만을 제공하며 확장된 접근조종기구들을 직접 제공하지 않는다. 그러나 Linux는 초보적으로 장치구동프로그람만이 아니라 파일체계와 같은 다른 구성부분들에 대해서도 동적으로 적재가능한 핵심부모듈들을 제공한다. 원리적으로 확장된 접근조종들을 각이한 보안모듈들에서 실현하여 핵심부모듈로서 리용할수 있다.

실천적으로 2.6이전의 핵심부는 핵심부모듈이 핵심부객체들에 대한 접근을 조정하도록 어떠한 하부구조도 제공하지 않기때문에 효과적인 보안모듈을 창조하는것이 문제로 되였다. 결과적으로 핵심부모듈들을 대체로 체계호출에 삽입하여 핵심부조작을 조종하는데 이것은 접근조종을 제공하는 방법으로서는 치명적인 제한을 가지고있다. 더우기 이러한 핵심부모듈들은 흔히 선택된 핵심부기능을 다시 실장할것을 요구하거나 핵심부에 모듈을 제공하는 패치를 요구하는데 이것은 모듈구성상 불합리하다. 그때문에 많은 보안프로젝트들이 핵심부패치들과 같은 Linux핵심부에 대한 확장된 접근조종프레임워크들이나 모형들을 설치하였다.

이러한 속에서 Linux보안모듈프로젝트는 많은 각이한 접근조종모형들을 적재할수 있으며 핵심부모듈들로서 실현될수 있는 Linux핵심부용 가볍고 일반목적인 접근조종프레임워크를 개발하였다. POSIX.1e자격(capability)들과 SELinux, 도메인과 형시행(DTE)을 포함하여 여러개의 확장된 접근조종실체들이 이미 LSM프레임워크를 리용하고있다.

LSM프레임워크는 Linux핵심부에 최소한으로 영향을 준다면 꼭같은 Linux핵심부를 가진 각이한 보안모형들을 실현할수 있다는 우점을 가지고있다. 이 일반성으로부터 확장된 접근조종들이 핵심부패치를 요구하지 않고 효과적으로 실장될수 있게 된다. LSM은 또한 POSIX.1e자격들의 현존보안기능을 기초적인 핵심부와 명백히 분리한다. 이것은 매몰형체계개발자들과 같은 전문화된 요구를 가진 사용자들로 하여금 성능의 견지에서 보안특징들을 최소한으로 감소하도록 한다. 또한 그것은 POSIX.1e자격들을 기초적인 핵심부로부터 보다 더 독자적으로 개발할수 있도록 한다.

보안프레임워크는 먼저 각이한 Linux보안프로젝트들의 현재의 보안기능을 적재가능한 핵심부모듈로서 어느 정도 충분히 실현하도록 하는데 중점을 두었다. 새로운 모듈을 설치하여도 보안을 제공하는데서 손실이 일어나지 말아야 하며 추가적으로 약간의 처리시간(overhead)은 가질수 있다.

이러한 대부분의 보안프로젝트들에서 핵심적인 기능은 접근조종이다. 그러나 일부 보안프로젝트들은 또한 보안검사기록(auditing)이나 가상화된 환경과 같은 다른 종류의 보안기능을 요구하였다. 더우기 접근조종에 대한 유연성측면에서 차이가 있다. 대부분의 보안프로젝트들은 단지 앞으로의 제한된 접근에 주목하였다. 즉 보통 현존Linux의 임의접근조종(DAC)론리에 의해 허가되는 접근을 거부할수 있다. 그러나 일부프로젝트들은 보통 현존DAC론리에 의해 거부된 접근들을 허가할것을 요구하였다. 이 일부등급의 허가(permissive)동작을 실현하자면 자격론리를 제공하는 모듈이 요구되였다. 일부보안프로젝트들은 DAC론리를 보안모듈로 옮기여 이것들이 그것을 대신하도록 하였다.

LSM문제는 Linux핵심부에 주는 영향을 최소화하는 한편 가능한한 많은 보안프로젝트들의 기능적인 요구를 통일하는것이다. 그러나 요구되는 특징들을 결합하는것은 높은 기능으로서 너무 지나치면 현재의 일반적인 Linux공동체에 접근할수 없다.

체계호출대면부는 접근을 조정하는데 알맞는 장소로서 사용자공간에 대한 추상화를 제공하여 핵심부와 상호작용한다. 사실 체계호출탐색표에서 입구점들을 덧쓰기하는데는 핵심부를 수정할 필요가 없으며 핵심부모듈을 리용하여 이 대면부를 조정하는것이 효률적이다.

LSM대면부의 기초적인 추상화는 내부핵심부객체에 대한 접근을 조정하는것이다. LSM은 모듈들이 표제 S가 내부핵심부객체 OBJ에 대하여 핵심부조작 OP를 수행할수 있는가? 라는 질문에 대답할수 있도록 한다.

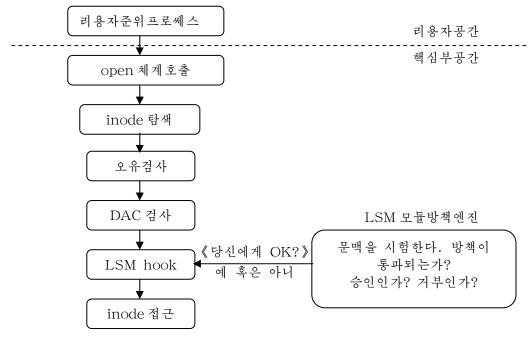



그림 7-6. LSM 후크기본방식

LSM은 모듈들이 그림 7-6에서 보여준것과 같이 접근의 바로 앞에서 핵심부코드에 후크들(hooks)을 두어 핵심부객체에 대한 접근을 조정하도록 한다. 핵심부가 내부객체에 접근하기 바로 앞에서 후크는 LSM모듈이 제공하는 함수에 대한 호출을 진행한다. 그 모듈은 접근이 허가하거나 혹은 접근을 거부하거나 오유코드를 강제로 되돌리도록 할 수 있다.

LSM프레임워크는 핵심부의 현존기구(mechanism)들을 리용하여 대체로 문자렬 혹은 단순화된 자료구조체들과 같은 리용자공급자료를 내부자료구조체로 번역한다. 이로하여 사용경쟁(TOCTTOU)시간과 비효과적인 반복탐색시간에 대한 검사시간이 줄어든다. 또한 LSM프레임워크가 핵심적인 핵심부자료구조체들에 대한 접근을 직접 조정하도록 한다. 이러한 방법으로 LSM프레임워크는 핵심부가 실제상 요구되는 봉사를 진행하기 전에 핵심부문맥에 대한 접근을 진행한다. 이것은 접근조종알갱이(granularity)를

개선한다.

POSIX.1e자격론리는 조잡한 준위의 알갱이에서 보통 거부되는 접근들을 허가할것을 요구한다. LSM은 이러한 론리를 제공하기 위해 이 허가적인(permissive)후크를 보안모듈로서 지원하는데 모듈은 핵심부가 거절하는 접근들을 허가할수 있다. 허가적인 후크들은 대체로 간단한 DAC검사와 결합되며 모듈이 DAC제한을 넘어가도록 한다. 그림 7-7에서는 리용자 ID검사가 허가적인 후크에 의해 넘겨질수 있는 리용자접근요구를 보여준다.

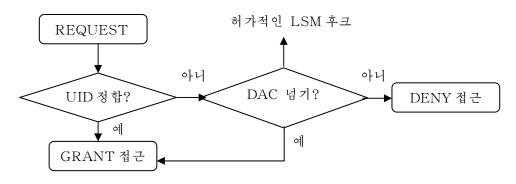



그림 7-7. 허가적인 LSM 후크.
(이 후크는 보안방책이 DAC 제한을 넘어가도록 한다.)

비록 LSM이 보안검사를 명백히 지원하도록 설계되지 않았다 하더라도 접근조종을 제공하는 특징들을 리용하여 일부형태의 검사기록을 지원할수 있다. 실례로 많은 현존 Linux보안프로젝트들은 접근검사의 기록을 자체의 접근조종들에 의해 진행한다. LSM도 역시 이러한 검사기록을 지원할수 있다. 일부보안검사기록은 또한 SNARE프로젝트에서와 같이 체계호출에 삽입하여 현존핵심부모듈들을 거쳐 지원될수 있다.

많은 보안모듈들에서 보안속성들을 핵심부객체들에 결합할것이 요구된다. 이것을 쉽게 하기 위해 LSM은 각이한 내부핵심부객체들에 불투명한 보안마당을 접속하여 지원한다. 모듈은 할당과 재할당, 동시성조종을 포함하여 이 마당들을 관리하는데 대한 책임을 진다.

마지막으로 모듈구성은 LSM설계에 대한 요구를 나타낸다. 한편으로는 명백히 보충적인 기능과 일부모듈들을 구성할데 대한 요구가 제기되고있다. 다른 한편 완전히 일반적인 보안방책구성은 취급하기 힘든것으로 알려져있다. 그러므로 LSM은 모듈탄창화를 허가하지만 대부분의 작업을 모듈 그자체에 밀어넣는다. 탄창화하려고 하는 모듈은 그자체가 LSM류형의 대면부를 수출(expert)하여야 하며 그 다음 적합한 때 적재된 모듈들을 호출한다. 적재된 첫번째 모듈은 모든 결정들에 대한 최종적인 조종을 가지며 언제 다른 모듈들을 호출하며 그 결과들을 어떻게 결합하는가를 결정한다.

제 2 절. 보안조작체계의 실현

1. Linux핵심부에서 보안기능의 실현

여기에서는 LSM핵심부기능의 실현에 대하여 서술한다.

LSM은 5개의 초보적인 방법들로 핵심부에서 실현된다. 첫째로, 불투명한 보안마당들을 어떤 핵심부자료구조체들에 추가한다. 둘째로, 핵심부코드내에서 각이한 점에서 보안후크함수들에 호출들을 삽입한다. 셋째로, 일반적인 보안체계호출을 서술한다. 넷째로, 핵심부모듈들이 자체를 보안모듈로서 등록하고 등록해제하도록 하는 함수들을 제공한다. 마지막으로 대부분의 자격론리를 선택적인 보안모듈에 옮긴다.

1) 불투명한 보안마당

불투명한 보안마당들은 void*지적자들로서 보안모듈들이 보안정보를 핵심부객체들과 결합할수 있게 한다. 표 7-1에서는 LSM핵심부패치와 해당 추상적인 객체에 의해 수정되는 핵심부자료구조체들을 보여준다.

표 7-1. LSM핵심부패치와 해당 추상적인 객체에 의해 수정되는 핵심부자료구조체들

구 조 체	객 체
task_struct	과제(프로쎄스)
linux_binprm	프로그람
super_block	파일체계
inode	관, 파일, 소케트
file	열린 파일
sk_buff	망완충기(파케트)
net_device	망장치
kern_ipc_perm	신호기, 공유기억기토막, 통보문대기렬
msg_msg	개별적인 통보문

이 보안마당들의 설정과 관련된 보안자료의 관리는 보안모듈에 의해 진행된다. LSM은 단지 요구되는 보안마당들을 관리하기 위해 모듈에 의해 실장될수 있는 보안후크들에 대한 호출마당들과 묶음(set)을 제공한다. 대부분의 객체들에 대하여 해당 핵심부자료구조체가 할당되고 해방될 때 보안모듈이 보안자료를 할당하고 해방하도록 하는데 alloc_security후크와 free_security후크를 정의한다. 다른 후크들은 필요할 때 보안자료를 갱신하도록 한다. 실례로 post_lookup후크는 탐색조작이 성공한 후에 inode에 대한 보안자료를 설정하는데 리용될수 있다. LSM은 보안마당들에 대한 잠그기를 제공하지 않는다. 이러한 잠그기는 보안모듈에 의해 진행된다.

여러 객체들이 보안모듈의 초기화전에 미리 존재하기때문에 모듈이 핵심부안으로 건립되였다면 보안모듈은 미리 존재하는 객체를 운전하여야 한다. 이 경우 여러 방법들을 리용할수 있다. 가장 간단한 방법은 이러한 객체들을 무시하는것인데 그것들을 모듈의 조종밖에 있는것으로 취급한다. 이 객체들은 그 다음 기초적인 Linux접근조종론리에 의해서만 조종된다. 두번째 방법은 모듈초기화기간 핵심부자료구조체들을 넘어가는것인데 이때에 미리 존재하는 모든 객체들의 보안마당들을 설정한다. 이 방법은 모든 객체들이 갱신되며(실례로 열린 파일은 프로쎄스가 접수하기를 기다리면서 UNIX도메인소케트상에 있을수 있다.) 적당한 잠그기기 진행된다는것을 담보하여야 한다. 세번째 방법은 리용할 때마다 미리 존재하는 객체들을 검사하는것이며 그 다음 필요한 때 미리 존재하는 객체들에 대한 보안마당을 설정하는것이다.

2) 보안후크함수들에 대한 호출들

앞에서 론의한것처럼 LSM은 핵심부객체들의 보안마당들을 관리하기 위해 보안후크들에 대한 호출묶음을 제공한다. 그것은 또한 이 객체들에 대한 접근을 조정하기 위해 보안후크들에 대한 호출묶음을 제공한다. 후크함수들의 이 두 묶음은 다 같이 대역적인 security_ops표에 있는 함수지적자를 통하여 호출된다.

```
struct security_operations {
    int (*ptrace) (struct task_struct * parent, struct task_struct *
child);
    int (*capget) (struct task_struct * target,
                   kernel_cap_t * effective,
                   kernel_cap_t * inheritable, kernel_cap_t *
permitted);
    int (*capset_check) (struct task_struct * target,
                         kernel_cap_t * effective,
                         kernel_cap_t * inheritable,
                         kernel_cap_t * permitted);
    void (*capset_set) (struct task_struct * target,
                        kernel_cap_t * effective,
                        kernel_cap_t * inheritable,
                        kernel_cap_t * permitted);
    int (*acct) (struct file * file);
    int (*sysctl) (struct ctl_table * table, int op);
    int (*capable) (struct task_struct * tsk, int cap);
    int (*quotactl) (int cmds, int type, int id, struct super_block *
```

```
sb);
        int (*quota_on) (struct dentry * dentry);
        int (*syslog) (int type);
        int (*settime) (struct timespec *ts, struct timezone *tz);
        int (*vm_enough_memory) (long pages);

        int (*bprm_alloc_security) (struct linux_binprm * bprm);
        void (*bprm_free_security) (struct linux_binprm * bprm);
        void (*bprm_apply_creds) (struct linux_binprm * bprm, int
unsafe);
        void (*bprm_post_apply_creds) (struct linux_binprm * bprm);
        int (*bprm_set_security) (struct linux_binprm * bprm);
        int (*bprm_check_security) (struct linux_binprm * bprm);
        int (*bprm_secureexec) (struct linux_binprm * bprm);

        int (*sb_alloc_security) (struct super_block * sb);
        void (*sb_free_security) (struct super_block * sb);
        int (*sb_copy_data)(struct file_system_type *type,
                        void *orig, void *copy);
        int (*sb_kern_mount) (struct super_block *sb, void *data);
        int (*sb_statfs) (struct super_block * sb);
        int (*sb_mount) (char *dev_name, struct nameidata * nd,
                        char *type, unsigned long flags, void *data);
        int (*sb_check_sb) (struct vfsmount * mnt, struct nameidata *
nd);
        int (*sb_umount) (struct vfsmount * mnt, int flags);
        void (*sb_umount_close) (struct vfsmount * mnt);
        void (*sb_umount_busy) (struct vfsmount * mnt);
        void (*sb_post_remount) (struct vfsmount * mnt,
                        unsigned long flags, void *data);
        void (*sb_post_mountroot) (void);
        void (*sb_post_addmount) (struct vfsmount * mnt,
                        struct nameidata * mountpoint_nd);
        int (*sb_pivotroot) (struct nameidata * old_nd,
                        struct nameidata * new_nd);
```

```
        void (*sb_post_pivotroot) (struct nameidata * old_nd,
                                    struct nameidata * new_nd);

        int (*inode_alloc_security) (struct inode *inode);
        void (*inode_free_security) (struct inode *inode);
        int (*inode_create) (struct inode *dir,
                              struct dentry *dentry, int mode);
        void (*inode_post_create) (struct inode *dir,
                                    struct dentry *dentry, int mode);
        int (*inode_link) (struct dentry *old_dentry,
                            struct     inode     *dir,     struct     dentry
*new_dentry);
        void (*inode_post_link) (struct dentry *old_dentry,
                                  struct    inode    *dir,    struct    dentry
*new_dentry);
        int (*inode_unlink) (struct inode *dir, struct dentry *dentry);
        int (*inode_symlink) (struct inode *dir,
                               struct    dentry    *dentry,    const    char
*old_name);
        void (*inode_post_symlink) (struct inode *dir,
                                     struct dentry *dentry,
                                     const char *old_name);
        int  (*inode_mkdir)  (struct  inode  *dir,  struct  dentry  *dentry,
int mode);
        void  (*inode_post_mkdir)  (struct  inode  *dir,  struct  dentry
*dentry,
                             int mode);
        int (*inode_rmdir) (struct inode *dir, struct dentry *dentry);
        int (*inode_mknod) (struct inode *dir, struct dentry *dentry,
                             int mode, dev_t dev);
        void  (*inode_post_mknod)  (struct  inode  *dir,  struct  dentry
*dentry,
                                    int mode, dev_t dev);
        int   (*inode_rename)   (struct   inode   *old_dir,   struct   dentry
*old_dentry,
```

```
                                struct    inode    *new_dir,    struct    dentry
*new_dentry);
        void (*inode_post_rename) (struct inode *old_dir,
                                      struct dentry *old_dentry,
                                      struct inode *new_dir,
                                      struct dentry *new_dentry);
        int (*inode_readlink) (struct dentry *dentry);
        int   (*inode_follow_link)   (struct   dentry   *dentry,   struct
nameidata *nd);
        int  (*inode_permission)  (struct  inode  *inode,  int  mask,  struct
nameidata *nd);
        int (*inode_setattr) (struct dentry *dentry, struct iattr *attr);
        int   (*inode_getattr)   (struct   vfsmount   *mnt,   struct   dentry
*dentry);
              void (*inode_delete) (struct inode *inode);
        int  (*inode_setxattr)  (struct  dentry  *dentry,  char  *name,  void
*value,
                                size_t size, int flags);
        void (*inode_post_setxattr) (struct dentry *dentry, char *name,
void *value,
                                     size_t size, int flags);
        int (*inode_getxattr) (struct dentry *dentry, char *name);
        int (*inode_listxattr) (struct dentry *dentry);
        int (*inode_removexattr) (struct dentry *dentry, char *name);
        int  (*inode_getsecurity)(struct  inode  *inode,  const  char  *name,
void *buffer, size_t size);
        int  (*inode_setsecurity)(struct  inode  *inode,  const  char  *name,
const void *value, size_t size, int flags);
        int   (*inode_listsecurity)(struct   inode   *inode,   char   *buffer,
size_t buffer_size);

        int (*file_permission) (struct file * file, int mask);
        int (*file_alloc_security) (struct file * file);
        void (*file_free_security) (struct file * file);
        int (*file_ioctl) (struct file * file, unsigned int cmd,
```

```
                        unsigned long arg);
        int (*file_mmap) (struct file * file,
                        unsigned long prot, unsigned long flags);
        int (*file_mprotect) (struct vm_area_struct * vma, unsigned
long prot);
        int (*file_lock) (struct file * file, unsigned int cmd);
        int (*file_fcntl) (struct file * file, unsigned int cmd,
                        unsigned long arg);
        int (*file_set_fowner) (struct file * file);
        int (*file_send_sigiotask) (struct task_struct * tsk,
                                struct fown_struct * fown, int sig);
        int (*file_receive) (struct file * file);

        int (*task_create) (unsigned long clone_flags);
        int (*task_alloc_security) (struct task_struct * p);
        void (*task_free_security) (struct task_struct * p);
        int (*task_setuid) (uid_t id0, uid_t id1, uid_t id2, int flags);
        int (*task_post_setuid) (uid_t old_ruid /* or fsuid */ ,
                                uid_t old_euid, uid_t old_suid, int flags);
        int (*task_setgid) (gid_t id0, gid_t id1, gid_t id2, int flags);
        int (*task_setpgid) (struct task_struct * p, pid_t pgid);
        int (*task_getpgid) (struct task_struct * p);
        int (*task_getsid) (struct task_struct * p);
        int (*task_setgroups) (struct group_info *group_info);
        int (*task_setnice) (struct task_struct * p, int nice);
        int (*task_setrlimit) (unsigned int resource, struct rlimit *
new_rlim);
        int (*task_setscheduler) (struct task_struct * p, int policy,
                                struct sched_param * lp);
        int (*task_getscheduler) (struct task_struct * p);
        int (*task_kill) (struct task_struct * p,
                        struct siginfo * info, int sig);
        int (*task_wait) (struct task_struct * p);
        int (*task_prctl) (int option, unsigned long arg2,
                        unsigned long arg3, unsigned long arg4,
```

```
                        unsigned long arg5);
        void (*task_reparent_to_init) (struct task_struct * p);
        void (*task_to_inode)(struct task_struct *p, struct inode
*inode);

        int (*ipc_permission) (struct kern_ipc_perm * ipcp, short flag);

        int (*msg_msg_alloc_security) (struct msg_msg * msg);
        void (*msg_msg_free_security) (struct msg_msg * msg);

        int (*msg_queue_alloc_security) (struct msg_queue * msq);
        void (*msg_queue_free_security) (struct msg_queue * msq);
        int (*msg_queue_associate) (struct msg_queue * msq, int
msqflg);
        int (*msg_queue_msgctl) (struct msg_queue * msq, int cmd);
        int (*msg_queue_msgsnd) (struct msg_queue * msq,
                              struct msg_msg * msg, int msqflg);
        int (*msg_queue_msgrcv) (struct msg_queue * msq,
                              struct msg_msg * msg,
                              struct task_struct * target,
                              long type, int mode);

        int (*shm_alloc_security) (struct shmid_kernel * shp);
        void (*shm_free_security) (struct shmid_kernel * shp);
        int (*shm_associate) (struct shmid_kernel * shp, int shmflg);
        int (*shm_shmctl) (struct shmid_kernel * shp, int cmd);
        int (*shm_shmat) (struct shmid_kernel * shp,
                      char __user *shmaddr, int shmflg);

        int (*sem_alloc_security) (struct sem_array * sma);
        void (*sem_free_security) (struct sem_array * sma);
        int (*sem_associate) (struct sem_array * sma, int semflg);
        int (*sem_semctl) (struct sem_array * sma, int cmd);
        int (*sem_semop) (struct sem_array * sma,
                      struct sembuf * sops, unsigned nsops, int
```

```
alter);

        int (*netlink_send) (struct sock * sk, struct sk_buff * skb);
        int (*netlink_recv) (struct sk_buff * skb);

        /* allow module stacking */
        int (*register_security) (const char *name,
                                  struct security_operations *ops);
        int (*unregister_security) (const char *name,
                                    struct security_operations *ops);

        void (*d_instantiate) (struct dentry *dentry, struct inode
*inode);

        int (*getprocattr)(struct task_struct *p, char *name, void
*value, size_t size);
        int (*setprocattr)(struct task_struct *p, char *name, void
*value, size_t size);

    #ifdef CONFIG_SECURITY_NETWORK
        int (*unix_stream_connect) (struct socket * sock,
                                    struct socket * other, struct sock *
newsk);
        int (*unix_may_send) (struct socket * sock, struct socket *
other);

        int (*socket_create) (int family, int type, int protocol, int
kern);
        void (*socket_post_create) (struct socket * sock, int family,
                                    int type, int protocol, int kern);
        int (*socket_bind) (struct socket * sock,
                            struct sockaddr * address, int addrlen);
        int (*socket_connect) (struct socket * sock,
                               struct sockaddr * address, int addrlen);
        int (*socket_listen) (struct socket * sock, int backlog);
```

```
        int (*socket_accept) (struct socket * sock, struct socket *
    newsock);
        void (*socket_post_accept) (struct socket * sock,
                                    struct socket * newsock);
        int (*socket_sendmsg) (struct socket * sock,
                               struct msghdr * msg, int size);
        int (*socket_recvmsg) (struct socket * sock,
                               struct msghdr * msg, int size, int flags);
        int (*socket_getsockname) (struct socket * sock);
        int (*socket_getpeername) (struct socket * sock);
        int (*socket_getsockopt) (struct socket * sock, int level, int
    optname);
        int (*socket_setsockopt) (struct socket * sock, int level, int
    optname);
        int (*socket_shutdown) (struct socket * sock, int how);
        int (*socket_sock_rcv_skb) (struct sock * sk, struct sk_buff *
    skb);
            int (*socket_getpeersec) (struct socket *sock, char __user
         *optval, int __user *optlen, unsigned len);
        int (*sk_alloc_security) (struct sock *sk, int family, int
    priority);
        void (*sk_free_security) (struct sock *sk);
      #endif    /* CONFIG_SECURITY_NETWORK */
      };

      extern struct security_operations *security_ops;
```

이 구조체는 그룹이 핵심부객체나 보조체계뿐아니라 체계조작에 대해 일부 높은 준위 후크들에 기초한 후크들과 련관된 보조구조체들의 묶음으로 구성되여있다. 매개 후크는 핵심부객체들과 파라메터들의 용어들에서 정의되며 관심은 리용자공간지적자들을 피하는데 둔다. 아래에서는 LSM핵심부코드에서 vfs_mkdir핵심부함수를 보여준다.

```
int vfs_mkdir(struct inode *dir, struct dentry *dentry, int mode)
{
  int error = may_create(dir, dentry, NULL);
```

```
        if (error)
                return error;

        if (!dir->i_op || !dir->i_op->mkdir)
                return -EPERM;

        mode &= (S_IRWXUGO|S_ISVTX);
        error = security_inode_mkdir(dir, dentry, mode);
        if (error)
                return error;

        DQUOT_INIT(dir);
        error = dir->i_op->mkdir(dir, dentry, mode);
        if (!error) {
                inode_dir_notify(dir, DN_CREATE);
                security_inode_post_mkdir(dir,dentry, mode);
        }
        return error;
    }
```

보안후크들은 굵은 문자로 표식하였다. 이 핵심부함수는 새로운 등록부들을 창조하는데 리용된다. 보안후크함수들에 대해 두개의 호출이 이 함수에 삽입된다. 첫번째 후크호출인 **security_inode_mkdir**는 새로운 등록부들을 창조하는 능력을 조종하는데 리용될수 있다. 후크가 오유상태를 되돌리면 새로운 등록부가 되돌려지며 오유상태는 호출자에게 전달된다. 두번째 호출인 **security_inode_post_mkdir**는 새로운 등록부의 inode 구조체에 대한 보안마당을 설정하는데 쓰일수 있다. 이 후크는 단지 보안모듈의 상태를 갱신할수 있으며 되돌이상태에 영향을 주지 않는다.

LSM이 또한 후크호출을 Linux핵심부 permission함수에 삽입한다하더라도 permission후크로는 파일생성조작을 조종하는데 불충분하다. 왜냐하면 조작의 형과 새로운 파일의 이름과 방식과 같은 잠재적으로 중요한 정보가 부족하기때문이다. 류사하게 Linux핵심부 may_create함수에 후크호출을 삽입하는것은 조작의 형과 방식에 대한 명백한 정보가 여전히 부족하기때문에 불충분하다. 그렇기때문에 후크가 해당 inode조작과 같은 대면부와 함께 삽입된다.

이 두개의 후크들을 vfs_mkdir에 삽입하기 위해 dir->i_op->mkdir호출에 끼워 넣

는다. 내부핵심부대면부들에 끼워넣으면 일부 LSM후크들에 대하여 동등한 기능을 제공한다. 그러나 이렇게 삽입하면 더 일반적인 많은 기능을 핵심부모듈을 통하여 실장할수 있게 한다. 핵심부모듈들은 력사적으로 GPL과 다른 특허를 리용하도록 되여있기때문에 삽입에 기초한 수법은 Linux핵심부개발자들이 LSM을 받아들이는데서 도전에 부닥치게 한다.

3) 보안체계호출

LSM은 일반적인 보안체계호출들을 제공하여 보안용응용프로그람들에 대한 새로운 호출들을 실장하도록 한다. 비록 모듈들이 /proc파일체계을 통하여 혹은 새로운 가상파일체계형을 정의하여 정보와 조작들을 수출할수 있다 하더라도 이러한 수법은 일부 보안모듈들의 요구에 대하여 볼 때 불충분하다. 실례로 SELinux모듈은 여러개의 현존체계호출들의 확장된 형태들을 제공하여 응용프로그람들로 하여금 핵심부객체들과 조작들과 련관된 보안정보를 구별하거나 포함할수 있도록 한다.

security체계호출은 현존Linux socketcall체계호출 다음으로 다양하고 간단한 형태로 되여있다. 모듈은 체계호출의 실장을 정의하기때문에 파라메터들을 해석하여 요구하는것을 선택한다. 이 요소들은 모듈식별자, 호출식별자, 요소배렬과 같은 모듈들에 의해 해석되게 된다. 기정적으로 LSM은 그 파라메터들로서 sys_security후크를 간단히 호출하기 위해 sys_security입구점함수를 제공한다. 어떤 새로운 호출들을 제공하지 않는 보안모듈은 ENOSYS을 되돌리는 sys_security후크함수를 정의할수 있다. 새로운 호출들을 제공하려고 하는 대부분의 보안모듈들은 자기들의 호출실장을 이 후크함수에 둘수 있다.

일부경우에 LSM이 제공하는 입구점함수는 보안모듈에 대하여 불충분하다. 실례로 SELinux가 제공하는 새로운 호출들중의 하나는 탄창에 대한 등록기들에 접근하려고 한다. SELinux모듈은 그 자신의 입구점함수를 실장하여 이러한 접근을 하도록 하며 모듈초기화기간 체계호출표에서 이 함수로서 LSM입구점함수를 대신한다.

4) 보안모듈의 등록

LSM프레임워크는 전통적인 UNIX초사용자의미를 가진 빈 후크함수들의 묶음으로 핵심부의 순차적인 부트기간에 초기화된다. 보안모듈이 적재되면 register_security함수를 호출하여 LSM프레임워크로서 그자체를 등록하여야 한다.

```
int register_security(struct security_operations *ops)
{
   if (verify(ops)) {
          printk(KERN_DEBUG "%s could not verify "
                "security_operations structure.\n", __FUNCTION__);
          return -EINVAL;
```

```
        }

        if (security_ops != &dummy_security_ops)
                return -EAGAIN;

        security_ops = ops;

        return 0;
    }
```

이 함수는 대역적인 security_ops표를 설정하여 모듈의 후크함수지적자들을 참조하며 핵심부는 접근조종결정에 대한 보안모듈들에 호출한다. register_security함수는 앞서 적재된 모듈을 덧쓰기하지 않는다. 일단 보안모듈이 적재되면 그것은 그자체가 부리워지도록 하는지에 대한 방책결정으로 된다.

보안모듈이 부리워지면 unregister_security를 리용하여 프레임워크로서 등록해제하여야 한다.

```
    int unregister_security(struct security_operations *ops)
    {
        if (ops != security_ops) {
                printk(KERN_INFO "%s: trying to unregister "
                        "a security_opts structure that is not "
                        "registered, failing.\n", __FUNCTION__);
                return -EINVAL;
        }

        security_ops = &dummy_security_ops;

        return 0;
    }
```

이것은 단순히 기정후크로 후크함수들을 교체하며 따라서 체계는 여전히 보안을 위한 일부 기본적인 수단들을 가지게 된다. 모듈이 불투명한 마당들을 재설정하는 쓸데없는 작업을 한다면 기정후크함수들은 불투명한 보안마당들을 리용하지 않으며 따라서 체계의 보안은 손상되지 않는다.

방책의 일반적인 구성은 까다롭다. 제멋대로한 방책구성으로 하여 정의되지 않은 결과가 나타나면 정의된 결과로 되도록 구성할수 있는 보안모듈을 개발하는것이 가능하다. 프레임워크를 단순화하기 위해서 기정이나 등록된 모듈 즉 초기의 모듈중 어느 하나의 모듈만을 인식한다. 보안모듈은 mod_reg_security대면부를 리용하여 그자체를 직접 초기의 모듈로서 등록할수 있다.

```
int mod_reg_security(const char *name, struct security_operations *ops)
{
   if (verify(ops)) {
          printk(KERN_INFO "%s could not verify "
                 "security operations.\n", __FUNCTION__);
          return -EINVAL;
   }

   if (ops == security_ops) {
          printk(KERN_INFO "%s security operations "
                 "already registered.\n", __FUNCTION__);
          return -EINVAL;
   }

   return security_ops->register_security(name, ops);
}
```

이 등록은 초기의 모듈로서 조종되며 따라서 모듈탄창화를 허용하는지는 방책이 결정한다. 이 간단한 대면부로서 프레임워크에서 기초적인 탄창화를 복잡하지 않게 제공할수 있다.

4) 자격들

Linux핵심부는 현재 POSIX.1e자격들의 보조묶음을 지원한다. LSM프로젝트에 대한 요구들중 하나는 앞에서 언급한것처럼 이 기능을 선택적인 보안모듈로 옮기는것이다. POSIX.1e자격은 전통적인 초사용자특권을 구분하고 개별적인 프로쎄스들로 그것들을 지적하는 기구를 제공한다.

본질적으로 특권허가는 보통 거부된 접근을 허가하기때문에 접근조종의 허가적인 형태이다. 결과적으로 LSM프레임워크는 적어도 Linux자격실장과 같은 파립도로서 허가적인 대면부를 제공하여야 한다. LSM은 자격검사를 진행하는 현존capable대면부를 핵심부내에서 리용하지만 capable함수를 LSM후크용의 간단한 외피(wrapper)로 축소하여 어떤 요구되는 론리가 보안모듈에 실장되도록 한다. 이 수법은 LSM이 capable에

관한 여러개의(500이상) 현존핵심부호출들을 리용하며 핵심부에 대해 변화가 전파되는 것을 피하도록 한다. LSM은 또한 후크들을 정의하여 다른 형태의 자격검사와 자격계산에 대한 론리를 보안모듈내에서 교갑화한다.

```
int cap_capable (struct task_struct *tsk, int cap)
{
   /* Derived from include/linux/sched.h:capable. */
   if (cap_raised(tsk->cap_effective, cap))
          return 0;
   return -EPERM;
}
```

간단한 비트벡토르인 프로쎄스자격묶음은 task_struct구조체에 보관된다. LSM은 불투명한 보안마당을 task_struct와 마당을 관리하는 후크들에 추가하기때문에 현존비트백토르를 그 마당으로 옮기는것이 가능해진다. 이러한 변화는 LSM프레임워크에서는 론리적이지만 다른 모듈들과 함께 탄창화를 쉽게 하기 위해 실장되지 않는다. LSM프레임워크에서 보안모듈을 탄창화하는 난관중 하나는 불투명한 보안마당들을 공유할데 대한 요구이다. 자격론리가 얼마동안 주류의 핵심부에로 통합되고 named와 sendmail과 같은 일부응용프로그람들이 그에 의존하기때문에 많은 보안모듈들을 자격모듈로서 탄창화하려고 한다. task_struct에 자격비트벡토르를 남기면 그것을 리용할 필요가 없는 쓸모없는 모듈들의 공간측면에서 이러한 구성이 쉬워진다.

자격들에 대한 Linux핵심부지원기능에는 또한 capset와 capget와 같은 두개의 체계호출이 포함된다. 현존응용프로그람들과의 호환성을 유지하기 위해 LSM은 이 체계호출들은 유지하지만 이 함수들에 대한 핵심부자격론리는 LSM후크들에 대한 호출에 의해 교체되였다. 결국 이 호출들은 security체계호출을 통하여 재실장되여야 한다. 이 변화는 자격용으로 적합한 대면부가 이 호출들을 직접 리용하기 보다는 libcap서고를 통하여 리용하기때문에 응용프로그람에 작은 영향을 주어야 한다.

LSM프로젝트는 자격보안모듈을 개발하였으며 많은 핵심부자격론리를 그안에서 조정한다. 그러나 핵심부는 여전히 이미 존재하는 Linux자격들의 흔적을 보여준다. task_struct로부터 불투명한 보안마당들에로 비트벡토르들을 옮기고 채계호출대면부를 재지정하면 자격모듈에는 완전히 표준적인 기본단계들만이 남게 된다.

5) 과제후크들

LSM은 보안모듈들이 프로쎄스보안정보를 관리할수 있으며 프로쎄스조작을 조종할수 있는 한 묶음의 과제후크들을 제공한다. 모듈들은 task_struct구조체의 보안마당을 리용하여 프로쎄스보안정보를 유지할수 있다. 과제후크들은 kill과 같은 프로쎄스간조작들에 대한 조종뿐아니라 setuid와 같은 현행프로쎄스에 대한 특권적인 조작들에 대한

조종을 지원한다. 과제후크들은 또한 setrlimit와 nice와 같은 자원관리조작들에 대한 미립도의 조종을 제공한다.

6) 프로그람적재후크

Linux자격들과 DTE, SELinux, SubDomain을 포함하여 많은 보안모듈들은 새로운 프로그람이 실행될 때 특권을 변화시킬수 있도록 할것을 요구한다. 결과적으로 LSM은 execve조작을 처리하는 기간에 한계점이라고 불리우는 한조의 프로그람적재후크를 제공한다. linux_binprm구조체의 보안마당은 모듈들이 프로그람이 적재되는 동안 보안정보를 유지하게 한다.

```
struct linux_binprm{
    char buf[BINPRM_BUF_SIZE];
    struct page *page[MAX_ARG_PAGES];
    struct mm_struct *mm;
    unsigned long p; /* current top of mem */
    int sh_bang;
    struct file * file;
    int e_uid, e_gid;
    kernel_cap_t cap_inheritable, cap_permitted, cap_effective;
    void *security;
    int argc, envc;
    char * filename;    /* Name of binary as seen by procps */
    char * interp;              /* Name of the binary really executed. Mos
t
                                    of the time same as filename, but could be
                                    different for binfmt_{misc,script} */
    unsigned interp_flags;
    unsigned interp_data;
    unsigned long loader, exec;
};
```

첫번째 후크는 보안모듈들이 이 보안정보를 초기화하도록 하고 프로그람을 적재하는것보다 먼저 접근조종을 수행하도록 하며 두번째 후크는 새로운 프로그람이 성과적으로 적재된 후에 모듈들이 과제보안정보를 갱신하도록 한다. 이 후크들은 또한 프로그람실행후 상태에 대한 정보를 조종하는데 리용할수 있다. 실례로 열린파일서술자들을 다시 유효하게 하는것 등을 들수 있다.

7) IPC후크들

보안모듈은 보안정보를 관리하며 LSM IPC후크들을 리용하여 System V IPC에 대한 접근조종을 진행할수 있다. IPC객체자료구조체들은 일반적인 보조구조체인 kern_ipc_perm을 공유하며 이 보조구조체에 대한 지적자만을 허가속성을 검사하는 현존 ipcperms함수에 넘긴다. 이때문에 LSM은 보안마당을 이 공유된 구조체에 추가한다. 개별적인 통보문들에 대한 보안정보를 제공하기 위해 LSM은 또한 msg_msg구조체에 보안마당을 추가한다.

```
struct kern_ipc_perm
{
    spinlock_t      lock;
    int             deleted;
    key_t           key;
    uid_t           uid;
    gid_t           gid;
    uid_t           cuid;
    gid_t           cgid;
    mode_t                  mode;
    unsigned long           seq;
    void            *security;
};

struct msg_msg {
    struct list_head m_list;
    long   m_type;
    int m_ts;              /* message text size */
    struct msg_msgseg* next;
    void *security;
    /* the actual message follows immediately */
};
```

LSM이 후크를 현존ipcperms함수에 삽입하므로 보안모듈은 매개의 현존 LinuxIPC허가속성에 대한 검사를 진행할수 있다. 그러나 이 검사들은 일부보안모듈들에 대하여서는 충분하지 않음으로 LSM은 또한 후크들을 개별적인 IPC조작들에 삽입한다. 이 후크들은 조작의 형과 명시적인 요소들에 대한 더 상세한 정보를 제공한다. 그것

들은 또한 System V통보문대기렬을 통하여 보내진 개별적인 통보문들에 대한 세밀한 조종을 제공한다.

8) 파일체계후크들

파일조작에 대해서는 3개 묶음의 후크들 즉 파일체계후크들, inode후크들, 파일후크들이 정의되여있다. LSM은 보안마당을 super_block, inode, file과 같은 결합된 핵심부자료구조체들에 추가한다. 파일체계후크들은 보안모듈들이 태우기와 statfs와 같은 조작들을 조종할수 있도록 한다. 그것에 inode후크들을 삽입하여 현존 permission함수를 다루지만 LSM은 또한 여러 다른 inode후크들을 정의하여 개별적인 inode조작들에 대해 더 세밀한 조종을 하도록 한다. 일부 파일후크들은 보안모듈들이 read와 write와 같은 파일조작들에 대한 추가적인 검사를 진행하도록 한다. 후크는 또한 보안모듈들이 소케트IPC를 통하여 열린파일서술자들의 접수를 조종하도록 하기 위해 지원된다. 다른 파일후크들은 fcntl과 ioctl과 같은 조작에 대해 더 세밀한 조종을 제공한다.

inode와 super_block에 보안마당들을 두면 dentry와 vfsmount구조체들에 그것들을 두는것으로 된다. inode와 super_block구조체들은 실제적인 객체들에 대응하며 names와 namespaces와는 독립이다. dentry와 vfsmount구조체들은 해당 inode와 super_block에 대한 참조를 포함하며 개별적인 이름이나 이름공간들과 결합된다. 첫번째 쌍의 구조체들을 리용하면 객체별명에 의해 생기는 결과들을 피할수 있다. 이 구조체들이 또한 관들과 소케트들과 같은 비파일객체들을 표현하기때문에 이 구조체들을 리용하면 핵심부객체들의 코드범위가 더 많아진다. 이 자료구조체들은 또한 파일체계코드에서 임의의 점에서 쉽게 사용할수 있으며 한편 두번째 묶음의 구조체들은 흔히 사용할수 없다.

9) 망후크들

한묶음의 소케트후크들을 리용하여 망환경에 대한 응용프로그람층접근을 조정한다. 이 후크들은 모든 소케트체계호출들에 대해 끼워넣기를 하여 모든 소케트기초의 통신규약들에 대해 일괄적인 조정을 진행한다. 능동적인 리용자소케트들은 련관된 inode구조체를 가지고있기때문에 분리된 보안마당은 socket구조체나 낮은 준위의 sock구조체에 추가되지 않는다. 소케트후크들이 프로쎄스들과의 관련속에 일반적인 망통화량를 조정하게 하면 LSM은 핵심부의 망접근조종프레임워크를 확장할수 있다. 실례로 sock_rcv_skb후크는 경계내의 파케트가 련관된 리용자공간소케트에 대기하기에 앞서 그의 목적지응용프로그람에 관하여 조정되도록 한다. IPv4, UNIX도메인과 Netlink통신규약들에 대하여 추가적인 후크들이 실장되였다.

망자료는 sk_buff(소케트완충기)구조체에 의해 교갑화된 파케트에 있는 탄창을 통과한다. LSM은 sk_buff구조체에 보안마당을 추가하여 매개 파케트에 기초하여 망층을 통과하여 보안상태를 관리할수 있게 하였다. 한 묶음의 sk_buff후크들이 이 보안마당의 생명주기관리를 위해 제공된다.

하드웨어와 쏘프트웨어망장치들은 netdevice구조체에 의해 교갑화된다. 이 구조체에 보안마당을 추가하여 매개 장치에 기초하여 보안상태를 관리할수 있게 되였다.

10) 다른 후크들

LSM은 모듈후크들과 한조의 짝대기준위체계후크들이라는 두개의 추가적인 묶음의 후크들을 제공한다. 모듈후크들은 핵심부모듈들을 생성하고 초기화하고 삭제하는 핵심부조작들을 조종하는데 리용될수 있다. 체계후크들은 체계호스트이름의 설정과 I/O포구들에 대한 접근, 프로쎄스계정의 구성과 같은 체계조작들을 조종하는데 리용될수 있다. 현존Linux핵심부는 자격검사를 리용하여 이 많은 조작들에 대하여 일부조종을 제공하지만 그 검사들은 단지 각이한 조작들중에서 조잡한 구별을 진행하며 일부요소정보는 제공하지 않는다.

2. SELinux의 접근조종

보안OS의 가장 기본으로 되는 기능이 접근조종기능이다. 보안OS는 기초로 되는 OS의 접근조종부분을 개량하여 보안을 높이고있다. SELinux에서는 Linux핵심부의 권한검사부분(코드)을 개조하고 보다 엄격하게 권한을 설정할수 있게 하고있다.

보안 OS의 접근조종은 아래와 같은 3개의 요소로부터 설정하고있다.

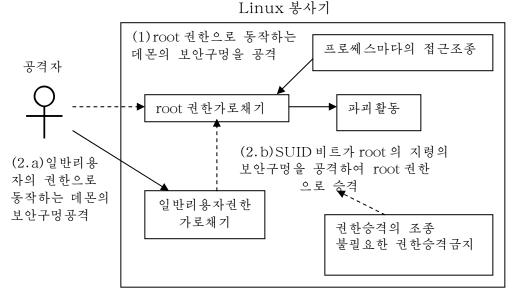



그림 7-8. SELinux의 보안기능

g. 강제접근조종(MAC)

Linux에서는 파일의 소유자가 그 파일의 권한을 설정하게 되여있다.

이것을 DAC(Discretionary Access Control:임의의 접근조종)라고 한다.

DAC에서는 파일의 소유자가 자유로 권한을 변경할수 있기때문에 체계전체의 보안설정을 일원적으로 관리하기 힘들다. 례하면 홈페지의 HTML파일의 소유권을 가진 리용자는 관리자만 제외하고 그 파일을 누구에도 읽기쓰기가능하게 설정할수 있다.

이에 대해 MAC에서는 자원의 접근조종에 관한 설정파일을 1개 준비하고 (이것을 보안방책파일이라고 한다.)그 설정을 변경할수 있는것은 보안관리자뿐이다.

따라서 MAC에서는 보안관리자는 접근조종에 관한 설정을 1개의 파일로 집중관리하고 그 설정을 기계전체에 철저하게 할수 있다.

h. **매 프로쎄스의 접근조종기능**

종래의 Linux에서는 root권한을 가진 프로쎄스는 모든 조작을 진행할수 있다.

이에 대해 SELinux에서는 root권한은 없으며 프로쎄스마다에 독자의 권한을 가지게 할수 있다. 실례로 아래그림에서는 Apache에는 홈페지에 대한 읽기허가밖에 주어져있지 않다. 마찬가지로 BIND에는 존(zone)파일에 대한 읽기허가밖에 주어져있지 않다. 그때문에 이것이외의 파일에는 호출할수 없다.

이에 의해 만일 그림의 (1)root로 침입을 허가하여도 보안 OS에서는 root권한이 존재하지 않기때문에 침입자는 한정된 권한밖에 가로챌수 없다.

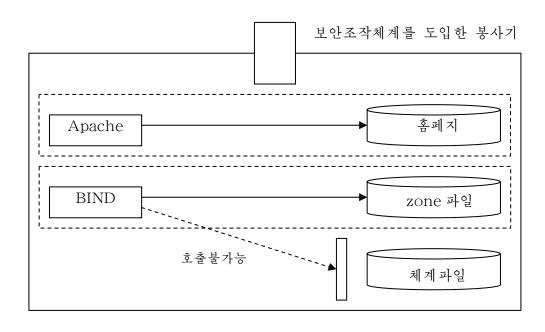



그림 7-9. 매개프로쎄스에 대한 접근조종실례

i. **권한승격의 조종**

Linux에서는 보통 SUID가 root로 된 프로그람을 통하여 일반리용자의 권한으로부터 root권한으로 승격할수 있게 된다. 또한 이러한 프로그람의 권한은 모든 리용자에

대하여 실행가능으로 되여있는것이 많기때문에 악용이 쉽다. 이에 대해서 보 OS에서는 불필요한 권한의 승격을 우의 그림에서처럼 일체 금지하고있다.

1) 보안방책

우선 접근조종에 대하여 보기로 하자. 접근조종의 참조모형의 구성요소들로서는 표제(subject)와 오브젝트(object), 규칙기지(RuleDB)로 되여있다.

표제는 실행프로그람과 같이 능동적인 실체를 나타내며 오브젝트는 파일과 포구, 기억기와 장치와 같은 자원과 정보를 담는 그릇을 말한다. 규칙기지는 실현되여야 하는 방책을 말한다. 표준적인 Linux에서 표제는 오브젝트에 대하여 접근시도를 할 때 규칙기지의 보안방책에 따라 그 접근여부를 묻는다. 이것이 참조유효성기구(RVM:Reference Validation Mechanism)에 의해 실현된다.

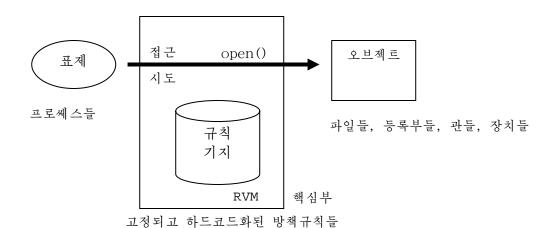
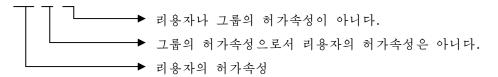



그림 7-10. 표준적인 리눅스핵심부의 접근조종

RVM은 고정되여있으며 핵심부에 실현되여있다. 표제는 프로세스를 나타내며 오브젝트는 파일과 등록부, 관, 장치를 나타낸다. 프로세스는 실제적이며 유효한 리용자와 그룹ID를 가진다.

파일체계에서 오브젝트는 inode로 관리하며 여기에 다음과 같은 접근방식이 있다.

rwx r-x --- uid gid

규칙기지는 고정되며 핵심부에서 하드코드화된다. 대체로 서술자생성보다는 열기에서 검사된다.

SELinux에서는 핵심부가 확장되여 보안방책이 동적으로 핵심부에 적재된다. 이에

따라 프로쎄스의 체계호출때에 접근조종이 진행되게 된다.

접근조종의 형태는 크게 3가지이다.

일반적인 접근조종

표제와 오브젝트는 보안속성을 가진다. 접근은 보안방책규칙에 기초하여 결정된다.

임의의 접근조종

리용자들은 요구에 따라 보안속성을 변화시킬수 있다. 리용자쪽에서 실행하는 프로그람은 접근규칙의 결과에 영향을 미칠수 있도록 되여있다.

강제접근조종

리용자들은 요구에 따라 보안속성들을 변화시킬수 없다. 리용자프로그람은 접근규칙의 강요내에서 작업하여야 한다. MAC접근규칙은 리용자가 아니라 조직에 의해 조종된다.

임의의 접근조종방법은 리용자와 프로그람들사이의 차이를 식별하기가 힘들다. 프로쎄스는 리용자대응물이며 임의의 코드를 실행할수 있다. 프로쎄스들은 접근조종속성들을 변화시킬수 없다. DAC는 일반적으로 친절한 쏘프트웨어환경을 요구한다. 리용자는 친절한 환경에서 마음대로 행동하게 된다.

결과 예견치 않았던 일이 생길수 있다. 우선 트로이목마와 같이 악의를 가진 쏘프트웨어를 열수 있다. 그리고 오유가 가능한한 최대의 특권에로 확대될수 있다. 지어 《믿는》리용자오유를 막지 못한다.

강제접근조종에는 다음과 같은 형태들이 있다.

벨-라파둘러(Bell-LaPadula)방책-다중준위보안

접근조종속성들에는 계층적인 보안준위, 비계층적인 범주들의 묶음이 있다.

고정규칙들은 《올리읽기금지, 내리쓰기금지》이다.

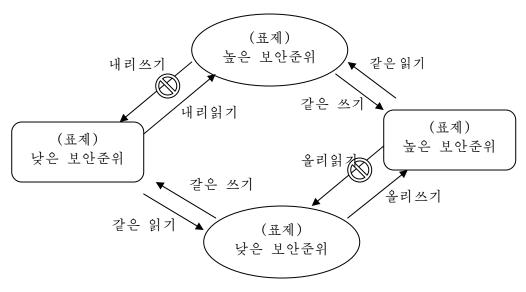



그림 7-11. 다중준위보안

비바(Biba)완전성모형

접근조종속성에는 계층적인 완전성준위, 비계층적인 완전성범주들의 묶음이 있다.

고정된규칙들은 《올리쓰기금지, 내리읽기금지》이다. 정확히 BLP/다중준위보안과 반대이다.

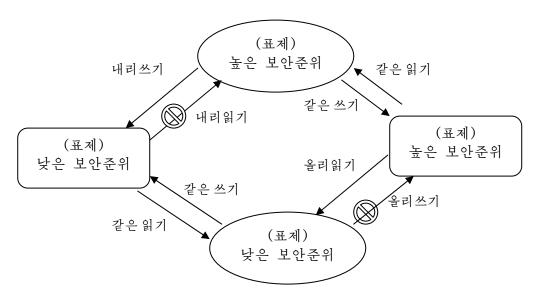



그림 7-12. 비바(Biba)완전성모형

BLP와 Biba는 공통적으로 MAC가 실현된다. 대표적으로 둘다 함께 실현되는데 엄밀하고 유연하지 못하다. 우의 MAC방책들외에도 상업적인 강제보안에 쓰이는 중국벽(Chinese Wall), 분리핵심부인 비간섭(non-interference), 일부역할에 기초한 접근조종방책들이 있다.

2) 보안문맥

SELinux는 표제와 오브젝트들에 대하여 보안문맥를 제공한다.

root:sysadm_r:sysadm_t

root: 리용자식별자

sysadm_r: 역할식별자

sysadm_t: 형식별자

보안문맥은 SELinux에서 단지 접근조종속성이다. 보안식별자(SID)는 핵심부내에서 능동인 보안문맥를 나타내는 수자이다. 원래 표준Linux에서는 표제(프로쎄스)접근조종속성들로서 실제적이고 유효한 리용자와 그룹ID들을 리용하며 오브젝트접근조종속성들로서 파일접근방식(rwx r-x r-x)와 리용자와 그룹ID들을 리용한다. SELinux에서는 프로쎄스의 형을 또한 《도메인》이라고도 한다. 역할(role)은 도메인형들을 련관된 그룹들로 결합하여 그것들을 리용자들로 나타낸다.

리용자 ——▶ 역할 ——▶ 도메인형 ——▶ 오브젝트형

보안문맥를 보려면 다음의 지령을 리용한다. 현재의 쉘에서 id지령의 뒤에 —context(-Z)를 추가하여 보안문맥를 확인해볼수 있다.

파일관련지령 ls에는 --lcontext, --context(-Z), --scontext를 추가한다.

프로쎄스관련지령 ps에는 --context(-Z)를 추가한다.

파일보안문맥변경을 다음의 지령을 리용한다. chcon지령은 문맥로서 파일의 보안문맥를 변화시킨다.

실례로

chcon system_u:object_r:shadow_t /tmp/foo

라고 할수 있다. setfile지령은 체계설치를 위해 준비된것으로서 자료파일을 리용하여 전체 체계를 재표식(relabel)한다. install과 cp에는 뒤에 Z(context)를 붙인다.

새로운 문맥에서 프로그람의 실행은 다음과 같이 한다. 새로운 보안문맥에서 runcon지령은 프로그람을 실행한다. 전체 문맥를 변화시키거나 다음과 같은 항목으로 문맥의 부분을 변화시킨다.

-t : 현재의 형을 명시된 형으로 변화시킨다.

-r : 현재의 역할을 명시된 역할로 변화시킨다.

-u : 현재의 리용자를 명시된 리용자로 변화시킨다.

3) 형시행과 역할

형시행(Type Enforcement)접근조종은 표제형(도메인)과 오브젝트형사이에 접근을 서술하며 접근허가를 정의하는데 다음 4개의 요소를 리용한다.

원천형 아카(aka)도메인

목표형 접근이 허가된 오브젝트

오브젝트클라스 접근이 적용하는 클라스

허가속성 접근허가에 대한 속성

SELinux는 30개의 오브젝트클라스들을 정의한다.

security, process, system, capability, filesystem,
file, dir, fd, lnk_ file,
chr_ file, blk_file, sock_file, fifo_file,
socket, tcp_socket, udp_socket, msgq, sem,
msg, shm, ipc, node, netif,
netlink_ socket, packet_socket, key_socket, rawip_socket, unix
_stream_socket, unix_dgram_socket, passwd

그것들 자체마다에는 더 엄밀히 나뉘여진 허가속성들이 있다. 실례로 file오브젝트클라스는 19개 허가속성을 가지고있다.

ioctl read write create getattr

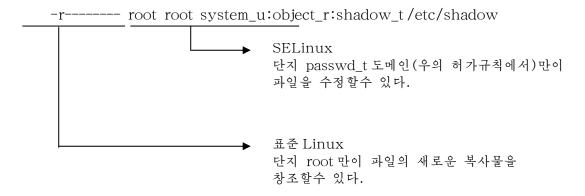
setattr lock relabelfrom relabelto append
unlink link rename execute swapon
quotaon mounton execute_no_trans entrypoint

간단한 실례로 형시행(TE)상태를 보기로 하자.

allow user_t bin_t : file { read execute };

user_t는 도메인형이며 bin_t는 오브젝트형, file은 오브젝트클라스를 나타낸다. 그리고 read execute는 허가속성이다. 이것은 형 user_t를 가진 프로세스는 형 bin_t를 가진 file들을 읽거나 실행할수 있다는 것을 나타낸다. 아래에서는 passwd프로그람을 실례를 들어 고찰한다.

allow passwd_t shadow_t : file
{ create ioctl read getattr lock write setattr append link unlink rename } ;

passwd_t도메인형을 가진 프로세스들은 shadow_t형을 가진 파일들에 읽기, 쓰기, 창조접근을 할수 있다. passwd프로그람은 passwd_t형으로 실행하는데 그것은 그림자통행암호파일(/etc/shadow)을 변화시킬수 있다. 그림자통행암호파일의 속성은 다음과 같다.

Linux체계의 리용자들은 체계를 리용할 때 아무 때나 자기의 통행암호를 변화시킬수 있어야 한다. 즉 uid와 euid가 pak이라는 가입자는 도메인형이 user_t이며 bash파일을 다룬다. 이때 통행암호를 변경하려면 euid가 root이고 도메인형이 passwd_t인 passwd파일을 통해서 도메인형이 shadow_t인 /etc/shadow파일에 대하여 창조와 쓰기속성을 가지고 통행암호를 변경할수 있어야 한다. 이와 같이 통행암호를 변경시키는 경우를 비롯하여 많은 경우 도메인이행문제가 제기된다.

그러면 표준Linux에서는 이 문제를 어떻게 해결하는가를 먼저 고찰하자.

표준Linux에서는 가입자 pak이 fork()체계호출을 통하여 passwd를 호출하면 passwd의 유효식별자 euid는 pak으로 된다. 이 passwd파일의 속성은

r-s--x--x root root

인데 여기서 x는 이 파일을 누구나 실행할수 있다는것을 나타내므로 pak은 /usr/bin/passwd를 실행할수 있다. 그러나 uid와 euid가 pak인 passwd프로세스는 속성이 r - - - - - - - - - root root 로서 아무속성도 허가하지 않는 /etc/shadow파일에 대하여 창조나 쓰기를 진행할수 없게 되여있다. 표준Linux체계에서는 s비트(set uid)가 passwd파일에 설정되여있어 일반리용자도 root권한을 가지고 shadow파일에 수정을 할수 있게 되여있다.

SELinux에서는 보안방책에 이러한 도메인이행을 서술하여 통행암호의 변경을 가능하게 하고있다.

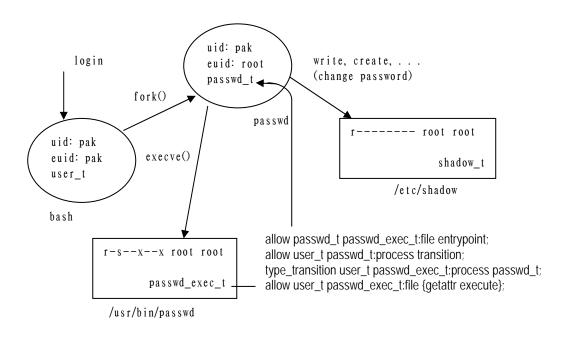



그림 7-13. SELinux에서 통행암호의 변경과정

역할은 도메인들과 리용자들을 결합하며 보다 더 구속적인 프로세스형이행을 진행한다. 프로세스형은 오직 역할정의에 의해 허가된다면 지어 형시행이 그것을 허가한다면 허가된다.

역할정의구문은 아래와 같다.

role user_r types passwd_t;

user_r: 허가된 형에 대한 역할

passwd_t: 허가된 형

쉘보안문맥변경을 다음의 지령으로 한다. newrole지령은 역할들과 도메인형을 변화시켜 새로운 쉘을 창조한다.

newrole -r role -t type [--args]

현재의 리용자의 명확한 재인증을 요구한다. 리용자의 특권준위를 변화시킬것을 요구한다. 기정값은 /etc/security/default_type로부터 얻을수 있다.

새로운 문맥은 방책에 의하여 허가되여야 한다. 유효한 보안문맥이여야 하며 도메인형이 새로운 형에 대한 접근으로 이행하여야 한다. 역할이 새로운 역할로 변화되는것이 허가되여야 한다.

4) 동작방식과 방책관리

SELinux동작방식은 허가방식(Permissive)과 시행방식(Enforcing)으로 갈라볼수 있다. 허가방식에서는 접근거부는 하지 않지만 거부를 로그기록하는데 방책개발기간에 리용된다. 시행방식은 접근거부를 시행하며 제품전개에 리용된다.

getenforce지령은 현재의 방식을 표시하며 setenforce지령은 방식을 절환한다.

허가방식을 불가능으로 하기 위해서는 CONFIG_SECURITY_SELINUX_DEVELOP가 없이 핵심부를 콤파일하여 허가방식을 불허한다.

설치된 방책파일의 위치는 다음과 같다.

/etc/security/selinux/policy.[ver] - 핵심부에 적재되는 2진방책파일

/etc/security/selinux/src/policy/policy.conf - 완전한 방책원천코드

/etc/security/selinux/src/policy policy.conf을 건립하는데 리용되는 원천파일

방책관리도구에는 다음과 같은것들이 있다.

sestatus

현재의 체계에 대한 기초적인 정보

seinfo

지령행방책해석

sepcut

기초적인 방책원천등록부열람기 및 편집기

apol

policy.conf에 대한 방책해석도구

seaudit

검사통보문조종과 해석

5) 오브젝트클라스

오브젝트클라스들은 다음과 같은데 정의되여있다.

/etc/security/selinux/src/policy/flask/security_classes

일부정의는 핵심부내에 건립되여있다. 오브젝트클라스는 다음과 같은 형태로 정의한다.

class file

여기서 file은 단순히 클라스식별자를 정의한다. 방책쓰기프로그람은 가끔

security_classes를 변화시킨다. 이것은 오브젝트클라스들이 핵심부나 리용자방식에서 오브젝트관리자를 변화시킬 때에만 변화된다.

매개 오브젝트클라스는 《접근벡토르》라고 하는 허가속성의 정의된 묶음을 가진다. 그리고 방책내의 허가속성정의들은 /etc/security/selinux/src/flask/access_vectors 들에 있다.

허가속성식별자들은 두가지 방법 즉 일반적인 상태(그룹과 같이 다중클라스들에 대해서 리용된다.)와 클라스상태#2(클라스상세허가속성)로 식별한다. 방책쓰기프로그람은 access_vectors를 변화시킨다.

일반적인 허가속성은 허가속성식별자들의 그룹을 정의하며 그룹과 같이 오브젝트클라스들을 결합시킨다.

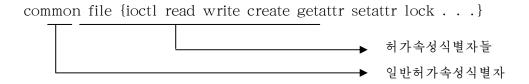



표 7-2. 파일오브젝트클라스의 허가속성들

№	허가속성이름	내 용
1	Read	파일내용의 읽기
2	Write	파일내용을 쓰거나 추가하기
3	Append	파일내용을 추가
4	Create	새로운 파일을 창조
5	Getattr	접근방식과 같이 파일속성을 읽기
6	Setattr	접근방식과 같이 파일속성을 변화시키기
7	Ioctl	ioctl체계호출은 다른 허가속성에 의해 주소화되지 않을것을 요구한다.
8	Unlink	하드련결의 제거
9	Link	파일에로의 하드련결을 창조
10	Lock	파일잠그기의 설정 과 해제
11	Rename	하드련결의 이름바꾸기
12	Relabelfrom	현존형에 기초하여 보안문맥을 변화시키기
13	Relabelto	새로운 형에 기초하여 보안문맥을 변화시키기
14	Mounton	Linux에서 등록부에 대해서만 의미를 가진다.

15	Swapon	파일이 페지화와 바꾸기공간에 대해서만 리용되게 한다.
16	Quoraon	분담(quota)을 가능으로 한다.
17	Execute	원래의 Linux실행과 같은 의미
18	execute_no_trans	도메인이행이 없이 파일을 실행하는 허가속성
19	Entrypoint	이 프로그람을 거쳐 새로운 도메인으로 들어가도록 하는 허가속성

표 7-3. 프로쎄스오브젝트클라스의 허가속성들

№	허가속성이름	내용
1	Transition	보안문맥를 변화시키기 위한 허가속성(낡은 도메인형과 새로운 도메인형에 대하여 검사한다.)
2	Fork	프로쎄스를 생성(fork)하거나 복제(clone)한다.
3	Sigchld	신호 SIGCHLD에 대한 허가속성
4	Sigkill	신호 SIGKILL에 대한 허가속성
5	Sigstop	신호 SIGSTOP에 대한 허가속성
6	Signull	보내진 신호가 없다.
7	Signal	다른 모든 신호
8	Ptrace	프로쎄스를 추적할수 있는 능력(실례로 오유수정 등)
9	getsched, setsched	프로쎄스우선도를 얻거나 설정한다.
10	Getsession	대화조종(session)정보를 얻는다.
11	getpgid, setpgid	프로쎄스그룹ID를 얻거나 설정한다.
12	getcap, setcap	자격을 얻거나 설정한다.
13	Share	상태공유를 허용한다.

6) 접근벡토르

TE접근벡토르규칙들의 문법은 다음과 같다.

rule_name src_types tgt_types : classes permissions ;

접근벡토르(AV)규칙들:

allow 형허가규칙

원천형이 목표형에 접근하는것을 허가하려면 기정으로 모든 접근허가는 없기때문에 오브젝트클라스와 허가속성에 대하여 작은 쪼각의 접근을 명시하여야 한다.

allow user_t bin_t : file {read getattr lock execute ioctl execute_no_trans};

이행 혹은 이행하지 않고 user_t도메인형이 bin_t파일들에 대해 읽기와 실행접근을 허가한다.

allow user_t self : process * ;

user_t도메인형들이 그자신에게 모두 접근하는것을 허가한다.

allow userdomain shell_exec_t : file {read getattr lock execute ioctl} ;

userdomain 속성을 가진 형들이 shell_exec_t파일들에 읽기/실행을 허가한다. 그러나 도메인이행만을 한다.(즉 exec_no_trans접근은 없다.)

neverallow　　불허가규칙

규칙이 방책이 콤파일되지 않았다면 어떤 불변방책을 침해하는것을 허가하지 않는다. 실시간체계에서는 포함되지 않는다. 이때에는 방책콤파일을 할 때 방책검사에 의해 시행되며 앞으로 실시간으로 넘어간다.

neverallow passwd_t ~{bin_t sbin_t ld_so_t} : file execute_no_trans ;

passwd_t도메인은 절대로 도메인이행이 없이는 bin_t, sbin_t, ld_so_t와는 다른 형을 가진 파일을 실행할수 없다.

neverallow domain ~domain : process transition ;

domain형은(《domain》은 속성) 새로운 형이 또한 domain형이 아니면 새로운 형으로 이행할수 있다.

auditallow접근이 허가될 때 일지기록접근이 TE허가되면 일지를 기록한다. 이것은 접근허가속성에 대하여 효과가 없다.

dontaudit　　접근이 거부될 때 일지를 기록하지 않음

접근이 거부될 때 검사를 하지 않는다. 기정으로는 검사가 거부로 되여있으며 기대되는 접근거부를 소거하는데 쓰인다.

형은 형시행의 기본으로서 보안문맥와 TE규칙에서 리용되며 《type》상태를 리용하여 정의된다.

형속성은 형의 그룹들을 결합하는 수단으로서 방책이 프로쎄스를 콤파일하는 기간에 쓰인다. 형속성은 TE규칙에서 보안문맥에서 아니라 형들의 위치에서 리용될수 있다. 형속성은 《attribute》와 《type》문구를 리용하여 정의한다.

형과 형속성은 같은 이름공간을 공유한다.

7) 형과 속성

형정의는 다음의 문법에 의하여 진행한다.

type *type_name* [alias *alias_name(s)*] [, *attrb1, ..., attribn*]

형과 별명들, 속성들은 공통적인 이름공간을 공유하며 유일하여야 한다. 별명과 속성들은 임의로 선택할수 있다. 별명들은 형이름과 동의어이다. 이것은 실행하는 체계에서 리용될수 있으며 실행하는 체계는 항상 초기의 형이름을 되돌린다. 속성들은 그 형을 다른 형들과 결합한다. 형은 여러개의 속성을 가질수도 있고 가지지 않을수도 있다.

속성정의는 현재의 판본에서 《attribute》문구를 통하여 정의된다.

attribute *attribu_name*

일단 정의되면 형정의에서 리용될수 있다. 속성들은 형문구를 통하여 형들로 쓰이며 《attribute》문구는 단지 이름을 보존한다.(방책의미를 포함하지 않는다.) 속성은 형그룹들에 규칙을 적용하기 위해 형들의 위치에 있는 TE규칙들에 리용될수 있다.

형정의실례를 보기로 하자. passwd프로그람을 제공하는 형들은 다음과 같이 정의한다.

type passwd_t, domain, privlog, auth, privowner ;

type passwd_exec_t, file_tpe, sysadmfile, exec_type ;

passwd_t, passwd_exec_t : 형이름

domain, privlog, auth, privowner, file_tpe, sysadmfile, exec_type : 지정된 속성들

속성들은 고유한 의미를 가지지 않는다. 그것들의 의미는 순전히 그것들을 리용하는 규칙에 기초하고있다.

형(원천과 목표):

《*》 는 모든 형을 의미한다.

《-》 는 속성의 형목록으로부터 형을 제거한다.

《~》 는 명시된 형 및 속성묶음의 보수에 리용된다.

하나이상의 식별자로 된것은 닫긴 괄호 《{}》에 렬거한다. 실례로

{type1_t type2_t typeN_t attribute   typeX_t}

classes

하나이상의 정의된 오브젝트클라스들은 《*》과 《~》을 리용할수 있다. 다중클라스들은 괄호 《{}》안에 렬거한다.

permissions

하나이상의 허가속성들은 명시된 클라스들에 대하여 정의된다. 모든 허가속성들은 명시된 모든 오브젝트클라스들에 대하여 유효하여야 한다. 《*》와 《~》가 리용될수 있다. 다중허가속성들은 괄호《{}》에 쌓인다. 다중규칙들이 같은 원천-목표-클라스를 명시하면 모든 허가속성들의 결합이 리용된다.

견본방책은 m4마크로들을 리용한다. 이것은 추상적인 개념을 리용하기가 더 쉬우며 SELinux방책언어에 대해서 본질적이지 않다. 공용마크로들은 ./policy/macros/*.te에 있다.

오브젝트클라스마크로의 실례:

file_class_set {file lnk_file sock_file fifo_file chr_file blk_file}

notdevfile_class_set {file lnk_file sock_file fifo_file}

여기서 필요없는 오브젝트들을 포함할수 있기때문에 주의하시오.

허가속성마크로실례:

rx_file_perms {read getattr lock execute ioctl}

r_dir_perms {read getattr lock search ioctl}

그러면 형이행규칙을 보기로 하자.

새로운 오브젝트에 대한 기정적인 형을 다음의 두가지 형식으로 명시할수 있다.

기정적인 프로쎄스이행과 새로운 파일오브젝트들에 대한 기정적인 형으로 할수 있다.

문법:

type_transition src_types tgt_types : class default_type ;

src_type & tgt_types : 《*》나 《~》, 형묶음을 리용할수 있다. 기정형은 단일형이다. 클라스는 어느 규칙이 형성되는가를 결정한다.

type_transition src_type tgt_type : process default_type ;

다른것이 요구되지 않으면 src_type를 가진 프로쎄스가 tgt_type로 된 파일을 실행할 때 프로쎄스는 default_type도메인을 가진다.

type_transition src_type tgt_types : file-related default_type ;

다른것이 요구되지 않으면 tgt_type의 등록부에서 src_type를 가진 프로쎄스가 새로운 파일관련오브젝트(실례로 file, dir)를 창조할 때 새로운 오브젝트는 default_type를 가진다.

type_transition userdomain passwd_exec_t : process passwd_t ;

기정으로 passwd_exec_t프로그람들을 실행할 때 userdomain속성을 가진 도메인들이 passwd_t에로 이행한다.

type_transition passwd_t mp_t :

{file lnk_file sock_file fifo_file} passwd_tmp_t ;

passwd_t프로쎄스가 tmp_t등록부(실레로 /tmp)에 새로운 파일체계오브젝트들을 창조할 때 그 새로운 파일들이 passwd_tmp_t형을 가진다. 일반적으로 도메인의 림시파일들을 보호하는데 리용하는 수법이다.

8) 제약과 보안문맥

추가적인 방책제약은 리용자들, 역할들 혹은 형들 그리고 오브젝트클라스와 허가속성사이의 관계를 고려하여 만족해야 하는 론리식으로 표현한다.

표준적인 방책제약은 /etc/security/selinux/src/policy/constraints에 있다.

제약상태문법은 다음과 같다.

```
# constrain class_set perm_set expression ;
#
# expression: (expression)
#  | not expression
#  | expression and expression
#  | expression or expression
#  | u1 op u2          u1와 u2은 리용자문맥(낡은것과 새것)
```

```
# | r1 role_op r2      r1와 r2은 역할문맥
# | t1 op t2          t1와 t2은 형문맥
# | u1 op user_set
# | u2 op user_set
# | r1 op role_set
# | r2 op role_set
# | t1 op type_set
# | t2 op type_set
# expression: 모두가 같은것과 같지 않은것
# op: == | !=                역할은 우선권(dominance)을 포함한다.
# role_op: == | != | eq | dom | domby | incomp
```

제약상태의 실례를 보자

constrain process transition (u1 == u2 or t1 == privuser or (t1 == crond_t and t2 == user_crond_domain));

여기서 u1 == u2는 낡은(u1)리용자와 새로운(u2)리용자가 같아야 한다는것;

t1 == privuser는 낡은 형(t1)이 《privuser》속성을 가져야 한다는것;

t1 == crond_t는 낡은 형(t1)이《crond_t》도메인형이라는것;

t2 == user_crond_domain는 새로운 형(t2)이 《user_crond_domain》속성을 가진다는것을 의미한다.

프로쎄스보안문맥이행에 대하여 제약은 다음과 같이 시행한다.

낡은 그리고 새로운 보안문맥에서 리용자동등성은 같아야 하거나 이행하는 프로쎄스의 도메인이 리용자를 변화시키는것보다 특권이 부여되며 이행하는 프로쎄스는 crond이고 새로운 도메인은 crond가 변화되는것을 허가한 crond리용자도메인들중의 하나이다.

constrain process transition (r1 == r2 or t1 == provrole) ;

프로쎄스보안문맥이행에 대한 제약은 다음과 같이 시행한다.

낡은 그리고 새로운 보안문맥들에서 역할은 같아야 하거나 혹은 이행하는 프로쎄스의 도메인은 역할을 변화시키는것보다 특권이 부여된다.

constrain dir_file_class_set {create relabelto relabelfrom}
  (u1 == u2 or t1 == privowner) ;

새로운 파일오브젝트들(dir_file_class_set)의 표식들에 대한 제약을 다음과 같이 시행한다.

새로운 오브젝트에 대한 리용자동등성은 프로쎄스에 대한 리용자와 같아야 하며 프로쎄스의 형은 파일소유자를 변화시키는것보다 특권이 부여된다.

초기의 SID보안문맥

초기의 SID들에 대한 보안문맥을 서술한다. 초기의 SID들은 미리 정의되며 체계초

기화나 미리 정의된 오브젝트들에 대해 리용한다.

문법:

sid sid_identifier security_context

sid_identifier는 미리 정의된 SID이름이다.

security_context는 user:role:type이다.

실례:

sid kernel system_u:system_r:kernel_t

sid security system_u:object_r:security_t

sid unlabeled system_u:object_r:unlabeled_t

파일체계에 대한 동작을 표식

태워질 때 동작을 표식하는 파일체계는 방책에 명시적으로 나타낼수 있다.

fs_use_* statements

genfscon statement

주어진 파일체계에 대한 방책에 없는 명세화를 표식한다면 파일체계와 그의 모든 오브젝트들은 《unlabeled》SID의 보안문맥으로 표식된다.

fs_use 상태들

실례들은 ./policy/fs_use에 있다.

fs_use_xattr fs_type 문맥은 selinux xattr를 지원하는 고전적인 파일체계 즉 현재에는 ext2, ext3, xfs에 대한것이다. 오브젝트문맥은 항상 파일체계에 포함된다.

fs_use_task fs_type문맥은 프로쎄스를 생성하는것과 같은것을 지적하는 오브젝트문맥이다. 문맥은 파일체계오브젝트표식이다.

fs_use_trans fs_type문맥은 프로쎄스를 생성하는것과 형이행규칙들에 기초한 오브젝트문맥이다. 문맥은 파일체계오브젝트표식이다.

genfscon상태

실례는 policy/genfs_contexts에 있다.

genfscon fs_type pathprefix [- file_type] context

pathprefix는 오브젝트이름들의 부분적인 경로이름이다.

file_type는 임의의 파일형구분자로서 b, c, d, p, l, s, -이다.

파일체계오브젝트문맥은 파일체계의 뿌리와 같다.

fs_use에 의해 조종되지 않는 파일체계형에 대해 쓰인다. 가장 긴 정합 pathprefix(그리고 임의의 파일형)는 오브젝트표식을 결정한다.

망오브젝트문맥

실례는 policy/net_contexts에 있다. 망오브젝트들에 대한 보안문맥표식규칙들은 port(tcp/udp포구들을 통한 통보문들), netif(망대면부), node(호스트주소)이다. 표식을 명시하지 않으면 초기의 SID가 문맥을 결정한다.

portcon protocol ports context

protocol은 tcp나 udp이며 ports context 포구번호(25) 또는 범위(137-139)로서 포구에로의 결합(binding)을 조종한다.

netifcon interface device_context packet_context

interface는 망대면부오브젝트이름(실례로 lo, eth0)이며 device_context는 netif오브젝트표식이고 packet_context는 수신된 파케트들의 기정표식이다.

nodecon ip_address ip_mask context

ip_address는 IP주소(실례로 192.168.0.0)이며 ip_mask는 IPv4망마스크(실례로 255.255.255.0)로서 모든 마디들의 보안문맥은 주소와 마스크에 의해 식별된다.

9) 방책파일

방책에는 론리와 조건이 리용된다. 론리는 방책에 정의된 변수들에 대해서 값이 true이거나 false이다. 값들은 실시간으로 변화될수 있다.

조건상태는 다음 3개의 부분으로 구성되여있다. 즉 조건식을 포함하는 《if》상태와 방책상태블로크, 임의의 《else》블로크로 되여있다.

방책에 대하여 제한된 실시간구성을 허가한다.

문법: bool name 기정값;

실례: bool user_ping false;

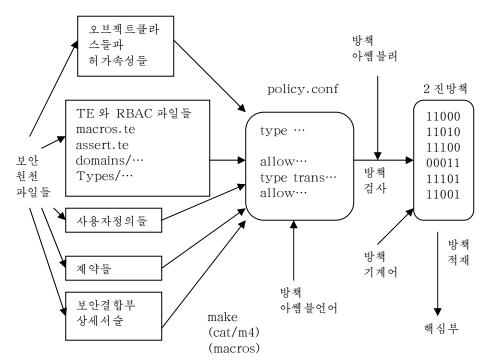



그림 7-14. 방책파일

기정으로 방책파일은 /etc/security/selinux/에 설치된다. 방책원천파일은 src/policy에, 2진방책을 건립하는데 쓰이는 파일은 src/policy/policy.conf이고 2진방책은 policy.[ver]이다.

초기의 Makefiles목표:

policy 검사와 개발을 위해 국부적으로 콤파일한다.

install 방책구성을 콤파일하고 설치한다.

load 방책구성을 콤파일하고 설치하고 적재한다.

reload 콤파일하고 설치하고 강제적재 및 재적재한다.

clean 림시파일들을 제거한다.

방책검사지령으로 policy.conf로부터 2진방책파일을 창조한다.

checkpolicy [-b] [-d] [-o 출구파일] [입구파일]

방책적재지령은 핵심부에 2진방책파일을 적재한다.

load_policy 경로

파일체계에 표식붙힘

디스크상의 파일체계들은 확장된 속성에 문맥들을 보관한다. 이것들이 fs_use_xattr 파일체계들이다.(ext2, ext3, xfs) 이것들은 초기시동이나 체계들사이를 옮겨갈 때 보안문맥들이 변하지 않도록 하는데 《security》라고 하는 확장된 속성의 이름공간을 리용한다.

파일들은 SELinux가 설치되는 동안 그리고 파일이 창조될 때 표식된다. 이 경우 fs_use동작에 따라 SELinux핵심부에 기초하여 실행하는 때 진행된다. 한편 파일들을 재표식하는 때에도 진행된다. 이때에는 chcon지령과 setfile지령, -Z항목과 함께 install, cp지령들을 리용한다.

방책해석과 오유수정

해석하는 자원으로는 방책원천파일들 policy.conf와 policy.XX, 방책검사오유통보문 /var/log/messages이다.

방책검사오유들은 방책콤파일오유들이다. 즉 1개 혹은 그이상의 방책문구들에서 문법 혹은 의미론적인 문제가 있다는것이다.

오유들은 policy.conf에 대하여 행번호참조와 문법오유들, 기호들을 통지한다.

검사통보문은 /var/log/messages에 기록된다. 일지기록파일입구점을 실례로 보기로 한다.

```
Jun 18 19:56:08 localhost kernel:audit(1087602968.172:0):avc:
   denied {read} for pid=16577 exe=/usr/bin/tail name=messages
   dev=sda2 ino=618992 scontext=root:staff_r:staff_t
   tcontext=system_u:object_r:var_log_t tclass=file
```

해석을 다음의 질문에 관하여 진행해나간다.

그것이 왜 발생하였는가?

정말 문제로 되는가?

거부사건이 있는가?

아니: 《dontaudit》를 리용한다.

예측되는 특권을 초과하는 문제인가?

예: 문제를 고정한다.

이것이 다른 프로그람과 고의적이지 않은 불일치인가?

예: 불일치를 해소한다.

혹시 목표가 표식되였는가?

아니: 파일을 재표식한다.

우리가 그 접근을 거부하려고 하였는가?

아니: 적당한 《allow》규칙을 고정한다.

3. SELinux보안방책작성

방책들은 사용자가 호출하는 역할(role)과 같은것들을 관리하는 규칙(rule)모임이다. 어떤 역할이 어떤 도메인(domain)을 신청(enter)할수 있는가와 어떤 도메인이 어느형에 접근(access)할수있는가를 규정한 모임이다.

당신은 어떤 체계를 설정하려는가 하는데 따라 방책파일(policy files)을 편집할수 있다. SELinux의 목적은 방책을 시행(enforce)하는것이다. 그래서 방책은 SELinux의 핵심부를 형성한다. 기정방책은 모든것을 거부하는것이고 모든 연산은 방책파일에서 명백하게 허용(permit)되여야 한다.

방책은 당신이 희망하는대로 당신의 체계를 융통성있게 구축하도록 해준다. 당신은 user_r와 sysadm_r 역할들을 호출할수 있는 사용자 A를 취하겠는가, user_r 역할만을 호출할수 있는 사용자 B를 취하겠는가를 선택할수 있다.

방책은 당신이 요구하는것만큼 엄밀하고 가볍게 될수 있다. 방책은 프로그람이 무엇을 동작시킬수 있는가, 어떻게 프로그람이 파일의 접근조종, 호상추적하는 프로그람, 송신신호 같은것들과 호상 작용할수 있는가를 조종할수 있다. 례를 들어 파일접근에 관해서 볼 때 어떤 도메인이 그것들을 창조하도록 당신은 /tmp안에 창조한 파일에 대한 방책을 가질수 있지만 다른 도메인은 그것을 호출할수 없다.

방책들은 그것이 리용되기 전에 2진형식으로 콤파일되여야 한다. 이것을 하기 위하여 우리는 방책등록부에서 make를 실행해야 한다. 그것은 Debian에서는 /etc/selinux/, RedHat에서는 /etc/security/src/policy이다. 한편 FedoraCore2에서는 /etc/security/selinux/src/policy(FedoraCore3에서는

/etc/selinux/src/policy이다)이다.

SELinux방책의 콤파일공정은 다음과 같다.

ㄱ) 방책구성파일은 하나로 련결되여야 한다.

방책구성파일은 .te로 끝나고 방책등록부와 그 안에 있는 부분등록부에 있다.

ㄴ) m4마크로처리기는 우의 련결의 결과에 응용된다. 그것은 다음에 policy.conf파일을 창조한다.

policy.conf파일은 방책등록부에 위치하고있으며 파일정의, 도메인정의 매 도메인이 할수 있는 규칙들, 역할들과 사용자들, 사용자가 어떤 역할들을 호출할수 있는가 하는 규칙들과 같은것을 포함한다.

ㄷ)계속하여 checkpolicy방책콤파일러가 Policy.VERSION파일을 창조한다. 거기서 VERSION은 판본번호이다. Policy.VERSION은 방책등록부에서 make install 지령을 실행하여 /etc/security/selinux등록부에 설치된다. 이 방책은 그 다음 재기동시에 적재된다.

그러나 만일 다시 방책을 runtime(실행시)변화를 하려고 한다면 실행핵심부에 새 방책을 적재하도록 방책파일등록부에서 make load지령을 실행할수 있다.

SELinux에서 어떤 연산을 집행하려고 할 때 그렇게 하려고 하는 사람의 능력은 그 사람의 보안문맥, 접근을 시도하려고 하는 오브젝트클라스(class of object), 그 오브젝트의 형(type)에 의해 결정된다. 실례로 파일은 파일의 클라스를 가지고있고 등록부는 등록부의 클라스를 가지고있으며 Unix도메인소케트(socket)는 sock_file의 클라스를 가지고있다. 접근을 시도하려는 오브젝트의 형이 방책에 의해 이미 결정되여있다. 권한이 없는 사용자 faye를 놓고 말해보자.

ls 1 /etc/shadow를 시도하려고 한다. 이것을 할 때 기록되는것은 다음과 같다. (sysadm_r로써 dmesg지령의 출력을 검사)

avc: denied { getattr } for pid=10387 exe=/bin/ls path=/etc/shadow dev=03:03 ino=129766

scontext=faye:user_r:user_t tcontext=system_u:object_r:shadow_t tclass=file

ls 1 /etc/shadow를 실행하는 사용자의 원천문맥(scontext)은 faye:user_r:user_t이다.

그래서 신분(identity)은 《faye》이며 user_t도메인에서 user_r역할을 취하고있다. 목표문맥(tcontext)은 system_u:object-r:shadow_t이다. 다른 말로 /etc/shadow는 형 shadow_t를 가지고있다. 목표클라스(tclass)는 파일이며 그래서 우리는 /etc/shadow가 형 shadow_t를 가지고있다는것을 알수 있다. 우의 전체적인 avc통보문은 우리가 시도해본것에 대한 상태를 보여주며 혹은 “getattr”/etc/shadow_t가 실패하였다는것을 보여준다.

그러면 /etc/shadow가 형 shadow_t를 가지고있다는것이 어디에 있는가?

그 대답은 등록부 file_contexts안에 있는 파일 file_contexts에 있다. 그것은 당신의 방책등록부안에 있다. 만일 우리가 이 파일안에서 /etc/shadow에 대해서 grep를 하면 다음의것을 보게 된다.

/etc/shadow.*   --   system_u:object_r:shadow_t

/etc/shadow의 형에 대한 명세서 역시 types.fc파일에 있는데 그것은 콤파일되서 file_contexts파일을 생성한다. 이 실례에서는 /etc/shadow.*를 호출한 파일은 system_u신분을 가지고있다. 왜냐하면 체계에 속해있는 임의의 파일들은 system_u를 가지고있기때문이다. object_r는 파일에 할당된 역할인데 그것은 역할과 같은 많은것들이 파일과 관계된다는것을 의미한다.

1) policy.conf, checkpolicy, the Makefile

SELinux는 일부측면에서 자체로 편집을 하게 되는 많은 구성파일을 가지고있다.

policy.conf

기본구성파일은 policy.conf이다. 이 파일은 모두가 련결되여있는 .te로 끝나는 파일들로 구성되여있다. 연산의 일반적인 과정에서 그것이 《make load》에 의해서 자동적으로 발생하기때문에 policy.conf파일을 편집하지 못하지만 만일 시험을 하면서 빨리 변경시키려면 편집할수 있다.

checkpolicy

이것은 방책콤파일러이며 《make reload》를 통해서 실행된다. Checkpolicy의 기본과제는 방책을 콤파일하는것이지만 그것 역시 방책을 물어보는데 사용할수 있다. 만일 user_t로써 checkpolicy를 실행하면 다음과 같은것을 볼수 있다.

```
faye@kaos:/etc/selinux$ checkpolicy
checkpolicy:  loading policy configuration from policy.conf
security:  4 users, 5 roles, 683 types
security:  29 classes, 71806 rules
checkpolicy:  policy configuration loaded
```

이것은 4개의 사용자를 가지고있다는것을 말해주며 현재의 etc/selinux/users파일에 faye, root, system_u, user_u이라는것이다.

여기서 user_r, sysadm_r, staff_r, system_r라는 5개의 역할(role)을 가지고있다. 어디에 5개의 역할이 있는가?

지령을 실행하면

```
grep ^role policy.conf | cut -f2 "-d "|sort -u
```

우에서 언급한 4개의 역할만을 가지고있다는것을 보여준다. 5번째 role object_r는 모든 파일에 할당된 역할이다. 그리고 명백하게 정의된다.(실지 방책에는 존재하지 않는

다.) 파일에 대한 역할이 관계없고 만일 파일의 보호준위를 요구한다면 특수형이 그 파일에 할당될것이라는것을 주의하시오.

우의 실례 역시 683형들과 29개 클라스들, 71806 규칙들을 가지고있다는것을 보여준다.

makefile

makefile은 아래와 같은 연산들을 제공한다.

install

Make install은 방책을 콤파일하고 설치하지만 그것을 적재하지는 않는다. 만일 SELinux핵심부를 실행하지 않고 다음번에 SELinux로 기동하도록 방책을 설치하려고 한다면 방책은 적재될것이다.

load

make load실행은 방책구성을 콤파일하고 설치, 적재한다. 체계를 재기동할 필요는 없다.

reload

make reload실행은 방책구성을 콤파일, 설치 그리고 적재하거나 재적재한다. makefile이 방책을 적재할 때 《load》라고 하는 기발파일은 방책원천등록부안에서 tmp등록부에 창조된다.

relabel

make relabel은 파일문맥구성에 기초한 파일체계를 재표식(relabel)한다. 파일문맥구성파일은 당신의 방책원천등록부에 있는 file_contexts에 있다.

policy

make policy는 시험 및 개발시에 국부적으로 방책을 콤파일한다. 이것은 방책을 콤파일하지만 실지로는 설치하지 않는다.

2) 속성들 : attrib.te파일

이 파일은 방책원천등록부안에 있으며 그것은 domain과 type들의 속성선언을 포함한다.

형속성은 류사한 성질을 가진 형들의 모임을 확인하는데 사용될수 있다. 매형은 임의의 수의 속성을 가질수 있으며 매 속성은 임의의 수의 형과 관련된다. 속성들이 형을 그룹화하고있는것처럼 도메인속성들은 도메인을 그룹화하고있다.

domian속성은 프로쎄스에 할당될수 있는 매형을 확인한다. 이 속성은 ps, top, inetd 등과 같은 실행될수 있는 모든 프로쎄스들과 관계된다.

privuser속성은 SELinux사용자신분을 변경할수 있는 모든 domain을 확인한다. 여기서는 표준 Unix id가 아닌 SELinux사용자신분에 대해 말하고있다는것을 념두에 둔다. 지령 grep ^type. *privuser policy.conf를 실행하면 그 신분을 변경할수 있는 도메인들이 sysadm_su_t, initrc_su_t, staff_su_t, run_init_t, local_login_t, remote_login_t, sshd_t, sshd_extern_t and xdm_t를 포함하고있다는것을 보여준다.

Privrole속성은 SELinux role을 변경할수 있는 모든 도메인을 확인한다. 도메인은 서로 다른 역할을 가진 프로쎄스를 만들수 있다. 실례로 newrole을 보자. 이 지령의 목적은 다른 역할로 변경하는것이다. privrole속성은 이것을 허가하기 위하여 newrole_t로 할당되여야 한다. Privrole은 다른 사용자역할들에 대한 변경을 허락한다. priv_system_role은 system_r에 대한 변경을 허락한다.

Privowner속성은 파일에 다른 SELinux사용자신분을 할당할수 있으며 프로쎄스 신분과 같지 않은 신분을 가진 파일을 창조할수 있는 모든 도메인을 확인한다. 실례로써 passwd_t를 사용하는것은 passwd_t프로쎄스가 그것을 실행하는 사용자의 신분을 가지고있으며 그것은 system_u 신원을 가지고 /etc/shadow를 재표식(relabel)하려고 하며 따라서 privowner를 요구한다.

userpty_type속성은 모든 user_devpts_t와 staff_devpts_t와 같은 비관리자적인 devpts형들을 확인한다. 실례로 지령

ls context /dev/pts를 실행하면 다음과 같은것을 볼수 있다.

```
crw-------  faye    staff    faye:object_r:staff_devpts_t     0
[snip]
```

여기서 /dev/pts/0은 형 staff_devpts_t를 가지고있다.

sysadmfile속성은 완전히 관리자로 접근할수 있는 파일에 할당된 모든 형들을 확인한다. shadow_t는 기정적으로 관리자로 접근할수 없다. 실례로 setfiles와 같은것들에 의해 접근할수 있다.

fs_type속성은 파일체계에 할당된 모든 속성들을 확인한다. security_t는 /selinux파일체계에 적용한다.

ptyfile속성은 ptys에 할당된 모든 형들을 확인한다. ttyfile속성 역시 여기서 응용한다.

xterm에서 ls context ‘tty’지령의 실행은 현재 접속하고있는 pty장치의 형을 보여준다.

례를 들면

```
faye@kaos:/etc/selinux$ ls --context `tty`
crw-------  faye    faye    faye:object_r:user_devpts_t      /dev/pts/1
```

다음 sysadm_r로 바꾸어서 같은 지령을 실행하면 다음과 같은것을 볼수 있다.

```
faye@kaos:/etc/selinux$ newrole -r sysadm_r
Authenticating faye.
Password:
faye@kaos:/etc/selinux$ id
uid=1000(faye) gid=1000(faye) groups=1000(faye),20(dialout),25(floppy),
```

29(audio),30(dip) \
context=faye:sysadm_r:sysadm_t
faye@kaos:/etc/selinux$ ls --context `tty`
crw------- faye faye faye:object_r:sysadm_devpts_t /dev/pts/1

pty가 현재 형 sysadm_devpts_t로 표식화되였다는것을 명심하시오.

login_contexts속성은 login형에 대한 기정문맥을 정의한 파일을 확인한다.(례 login, cron)

login형에 대한 기정문맥은 파일 /etc/security/default_contexts에 있다.

3) 사용자관련파일들

users 파일

이 파일은 배포물의 방책원천등록부에 있다. /etc/selinux/users에 있다. 그것은 SELinux체계가 인식하여야 하는 매 사용자에 대한 정의를 포함하고있다. 만일 사용자가 이 파일에 명백히 정의되여있다면 그 사용자신분은 그 보안문맥의 첫 부분에 서술된다. 보안문맥은 신분(identity), 역할(role), 도메인(domain), 형(type)으로 이루어져있다. 사용자는 자기 소유의 현재보안문맥을 id지령을 실행하여 검색할수 있다. 만일 사용자신분이 사용자파일에 정의되여있지 않다면 user_u신분을 할당한다.

다음과 같은것이 users파일에 들어있다고 하자.

user root roles { staff_r sysadm_r };

이것은 사용자신분이 root임을 정의하고 root가 staff_t와 sysadm_r역할로 들어가도록 한다. newrole지령은 사용자역할을 변경하는데 사용될수 있거나 혹은 콘솔에서 가입시 우의 역할중 하나로 들어가도록 선택할수도 있다.

user pak roles { staff_r sysadm_r };

우의 두가지 례는 root사용자가 sysadm_r를 가지고있어야 한다는것을 보여주며 또한 다른 사용자(pak)가 체계관리자역할로 접근할수 있다는것을 볼수 있다. 만일 사용자 root에 대한 역할정의가 sysadm_r를 가지고있지 않다면 pak은 root보다 더 강력하게 될것이다.

user kim roles { user_r };

사용자 kim은 user_r 역할만 접근할수 있다. 그것은 일반적으로 권한이 없는 사용자역할이다.

user system_u roles system_r;

system_u신분은 파일과 등록부, 소케트 등과 같은 프로세스나 오브젝트에 대한 사용자신분이다. 사용자는 사용자프로세스에 system_u신분을 할당하지 못한다. 왜냐면 system_u가 데몬을 위한것이기때문이다. 만일 사용자가 system_u를 가지고있다면 system_r나 임의의 데몬령역에 대한 접근을 진행한다.

user.te파일

이 파일은 부분등록부 domains에 있다. 이것은 현재의 체계에서 사용자에 대해 권한이 없는 도메인들을 포함한다. 이 파일에서 다음과 같은 행을 볼수 있다.

full_user_role(user)

이 행은 사용자역할로서 그 홈등록부에서 bin_t프로그람의 실행, 사용자의 홈등록부에 user_home_dir_t할당과 그 안에서 등록부에 user_home_t할당과 같은 표준조작들을 하도록 모든 조작을 가능하게 한다.

```
full_user_role(staff)
allow staff_t unpriv_userdomain:process signal_perms;
can_ps(staff_t, unpriv_userdomain)
allow staff_t { ttyfile ptyfile tty_device_t }:chr_file getattr;
```

이것은 령역 staff_t를 정의한다. 두번재 행은 staff_t 도메인이 user_t와 staff_t와 같은 권한이 없는 도메인으로 실행하는 프로쎄스에 신호를 보내도록 한다. 세번째 행은 staff_t가 ps를 실행하고 권한이 없는 사용자도메인으로 프로쎄스를 보게 해준다.

staff_t는 ps를 실행하고 user_t와 사용자도메인으로 되여있는 모든것을 볼수 있지만 user_t는 할수 없다.

4번째행은 staff_t가 임의의 말단장치속성에 접근하도록 한다.

dontaudit unpriv_userdomain sysadm_home_dir_t:dir { getattr search };

이 행은 ls l 혹은 cd /root지령과 같은것을 실행함으로써 혹은 /root등록부안에서 파일접근을 시도함으로써 /root등록부에 대해 접근하려는 권한이 없는 사용자(user_t와 같은)도메인을 확인하지 않는것을 말한다. 이 행은 《dontaudit source destination:destination class {what was attempted }》로써 읽을수도 있다.

```
# 역할 $1_r로부터 $2_r로 변경하고 tty를 정당히 재표식한다.
define(`role_tty_type_change', `
allow $1_r $2_r;
type_change $2_t $1_devpts_t:chr_file $2_devpts_t;
type_change $2_t $1_tty_device_t:chr_file $2_tty_device_t;
')
```

첫행은(설명문이 아닌) 마크로 role_tty_type_change를 정의한다. 이 마크로는 사용자가 역할을 변경하고 사용하고있는 tty를 재표식하도록 한다.(staff_r에서 sysadm_r을 얻으려고 newrole지령을 사용하여 현재의 tty를 변경할 때와 같이)

두번째 행은 $1_r가 $2_r에로 전환하는것을 허락하도록 한다.(여기서 $1_r는 staff_t일것이고 $2_r는 sysadm_r일것이다.)

tty의 재표식은 세번째와 네번째 행에 의해 진행된다.

```
ifdef(`newrole.te', `
#
# Allow the user roles to transition
# into each other.
role_tty_type_change(sysadm, user)
role_tty_type_change(staff, sysadm)
role_tty_type_change(sysadm, staff)
')
```

이 블로크는 만일 newrole.te가 정의되였다면 나타난 역할이 서로 전환이 가능하도록 하게 한다는것을 보여준다. 이 상태문은 이미 정의된것으로써 role_tty_type_change마크로의 호출이다.

user_macros.te파일

이 파일은 사용자등록가입도메인에 대한 마크로를 포함하고있다.

사용자등록가입도메인에 대한 마크로

```
# user_domain() is also called by the admin_domain() macro
undefine(`user_domain')
define(`user_domain',`
```

이것은 마크로 user_domain을 정의한다.

```
# Use capabilities
allow $1_t self:capability { setgid chown fowner };
dontaudit $1_t self:capability { sys_nice fsetid };
```

이 토막에서 《capabilities》는 그룹 id(setid)를 변경하는 가능성을 의미한다. 프로그람은 자기의 그룹 id를 변경하려고 할 때 setgid()를 호출할수 있다. 프로그람은 이 가능성이 허가되지 않았다면 setgid()를 호출할수 없다.구체적인 정보를 보려면 /usr/include/linux/capability.h를 보시오.

첫행은 $1_t 형이 setgid에 대한 자격을 허락한다. $1은 코드를 호출하는 첫 파라메터이다.(마크로를 호출한 코드)

두번째 행은 sys_nice(순서짜기우선권을 확장하는 능력) 혹은 fsetid(setuid와 setgid파일조종과 관련된)를 검사하지 않는다는것을 의미한다. 이것들은 몇번이고 요구되는 자격들이다.

```
# Type for home directory.
ifelse($1, sysadm, `
type $1_home_dir_t, file_type, sysadmfile, home_dir_type, home_type;
```

type $1_home_t, file_type, sysadmfile, home_type;

tmp_domain($1)

', `

type $1_home_dir_t, file_type, sysadmfile, home_dir_type, user_home_dir_type, home_

type, user_home_type;

type $1_home_t, file_type, sysadmfile, home_type, user_home_type;

우의 행들은 만일 $_1이 sysadm이면 첫 블로크를 집행하고 그렇지 않으면 두번째 블로크를 집행하라는것을 의미한다.

둘다 홈등록부와 tmp_domain(/tmp파일접근에 대한 마크로)에 대한 형을 정의한다. 그래서 첫 블로크는 sysadm stuff와 관련되며 두번째 블로크는 나머지것들과 관련된다.

do not allow privhome access to sysadm_home_dir_t

file_type_auto_trans(privhome, $1_home_dir_t, $1_home_t)

tmp_domain($1, `, user_tmpfile')

')

여기서 privhome도메인이 sysadm_home_dir_t를 호출하는것을 허가하지 않는다. 실례로 procmail을 들어보자.

procmail이 사용자의 홈등록부에서 파일을 창조하는 메일을 전송할 때 sysadm등록부(/root)로 되는것을 바라지 않는다.

《file_type_auto_trans》는 새 파일에 대한 기정의 형을 설정하는 방법이며 그것이 창조된 등록부와 같은 형이 된다.

일반적으로 파일을 창조할 때 그 파일은 그것이 창조된 등록부와 같은 형을 가진다. 창조시에 등록부에서 서로 다른 형을 가지도록 하는 방법은 두가지이다. 첫번째 방법은 open_source()를 리용한것이며 두번째는 file_type_auto_trans를 리용하는것이다.

첫번째 방법이 가능하려면 그것을 허용하는 file_type_trans규칙을 가져야 한다. 그 다음 file_type_autp_trans는 그것을 기정적으로 설치한다.

allow ptrace

can_ptrace($1_t, $1_t)

이것은 첫 파라메태($1_t)가 ptrace를 하게 한다. 혹은 두번째 파라메터의 처리를 추적한다. 이 행은 다음과 같이 $1_t가 자기의 ptrace를 허용하게 한다.

#홈등록부에서 파일을 창조하고 접근하고 제거한다.

file_type_auto_trans($1_t, $1_home_dir_t, $1_home_t)

allow $1_t $1_home_t:dir_file_class_set { relabelfrom relabelto };

여기서 도메인 $1_t는 형 $1_home_dir_t인 등록부안에서 파일을 창조한다. 그리고 기정으로 창조된 파일은 형이 $1_home_t이다. 두번째 행은 $1_t가 그밖의 아무것에 대해

서 형이 $1_home_t인 파일을 재표식하게 한다. 그리고 그 밖의 아무것으로부터 $1_home_t로 형을 바꾸게 한다.

Bind to a Unix domain socket in /tmp.

allow $1_t $1_tmp_t:unix_stream_socket name_bind;

이것은 $1_t가 Unix도메인소케트에 대한 결합(bind)을 허락한다. 그래서 그 다음에는 다른 프로쎄스로부터 접속을 받을수 있다.

$1_tmp_t는 tmp_domain의 형이다. Name_bind는 $1_t가 /tmp안에서 이름에 대한 결합을 하도록 허락한다.

초기사용자도메인에 대한 마크로들

아래에는 user_t와 관련된 내용이 있다.

undefine(`full_user_role')

define(`full_user_role', `

user_t/$1_t is an unprivileged users domain.

type $1_t, domain, userdomain, unpriv_userdomain, web_client_domain;

$1_r is authorized for $1_t for the initial login domain.

role $1_r types $1_t;

allow system_r $1_r;

Grant permissions within the domain.

general_domain_access($1_t);

마크로 full_user_role를 정의한다. 형 $1_t/user_t를 정의하고 거기에 목록화된 4가지 속성을 준다.

$1_r/user_r는 $1_t/user_를 가질수 있다. System_r는 다음$1_r/user_r에 대한 호출을 허락한다. general_domain_access는 $1_t가 $1_t에 있는 프로쎄스를 볼수 있게 허락한다. 또한 다른것들중 /proc/#에 있는 파일들을 볼수있게 한다.(파일 core_macros.te를 검사)

Read /etc.

allow $1_t etc_t:dir r_dir_perms;

allow $1_t etc_t:notdevfile_class_set r_file_perms;

allow $1_t etc_runtime_t:{ file lnk_file } r_file_perms;

user_t가 /etc_t(/etc/의 형)을 읽도록 허락한다. 리용자는 /etc안의 파일을 읽고 볼수 있으며 /etc에서 ls 1 command와 같은것들을 할수 있다. 만일 파일 core_macros.te를 주시해보면 notdevfile_class_set 가 파일, 기호련결, 소케트파일, fifo파일과 같은 비장치형파일클라스들과 련관되여있다는것을 알수 있다. etc_number_t는 /etc안에 있는 어떤파 일에 대한 형이다.(파일 file_contexts안에 있는 《etc_runtime_t》에 대한 grep)

```
undefine(`in_user_role')
define(`in_user_role', `
role user_r types $1;
role staff_r types $1;
')
```

마크로 in_user_role을 정의한다. 도메인은 임의의 사용자역할(role)로 사용될수 있다. 마크로는 관계가 있는 도메인의 .te파일에서 호출되여야 한다. 파일 passwd.te를 주의해보자.(passwd프로그람에 대한것) in_user_role마크로가 호출되며 그것을 통과하는 passwd_t 파라메터를 가지고있다.

```
role user_r types passwd_t;
role staff_r types passwd_t;
```

user_r와 staff_r가 passwd프로그람을 실행할수 있다는것을 의미한다는것을 알수 있다. 만일 새 역할을 추가하려면 여기에 in_user_role을 편집해야 한다.

4) 체계관련파일들

여기서는 sysadm_r 역할 즉 체계관리자와 관련한 방책을 보기로 한다. 앞에서 이미 SELinux신분이 sysadm_r을 위임받을수 있는가를 보았다.

admin_macros.te파일

이 파일은 체계관리자도메인에 대한 마크로를 포함한다.

```
undefine(`admin_domain')
define(`admin_domain',`
# Inherit rules for ordinary users.
user_domain($1)
```

마크로 admin_domain을 정의하고 그것이 user_t와 같은 규칙을 가지도록 한다. 이 경우에 $1은 sysadm일것이다.

```
allow $1_t policy_config_t:dir { getattr search };
allow $1_t policy_config_t:file getattr;
```

sysadm_t가 policy_config_t인 형을 가지고있는 등록부안에서 파일 및 등록부탐색이나 getattr(ls -l과 같은것)을 하도록 허락한다.

```
allow $1_t kernel_t:system syslog_read;
```

sysadm_t가 체계일지를 읽게 한다. kernel_t는 핵심부 그자체의 형이다. system은 연산의 클라스이다. 연산은 syslog를 읽으려고 한다.

```
# Use capabilities other than sys_module.
allow $1_t self:capability ~sys_module;
```

sysadm_t가 sys_module외에 모든 자격을 사용하도록 한다. 그것은 모듈적재에 사용된다.

```
# Get security policy decisions.
can_getsecurity($1_t)
```

만일 파일 core_macros.te를 주시해보고 can_getsecurity를 탐색하면 다음과 같은것을 볼수 있다.

```
# can_getsecurity(domain)
#
# Authorize a domain to get security policy decisions.
#
define(`can_getsecurity',`
allow $1 security_t:dir { read search getattr };
allow $1 security_t:file { getattr read write };
allow $1 security_t:security { check_context compute_av compute_create compute_relabel compute_user };
')
```

여기서 $1에 형 security_t(사용자의 방책 원천등록부)인 등록부에 대한 속성얻기 및 읽기, 탐색이 허가된다. $1은 형 security_t인 등록부에 있는 파일을 읽기 및 쓰기하고 속성을 얻을수 있다.

마지막으로 $1은 문맥유효성을 검색할수 있고 방책이 원천문맥이 목표문맥을 호출하게 하는가를 검사하며 새 오브젝트의 표식화에 대한 문맥을 계산하고 오브젝트를 재표식화할 때 새 문맥을 계산하고 어떤 사용자문맥이 주어진 원천문맥으로부터 영향을 받는가를 결정한다.

```
# Change system parameters.
can_sysctl($1_t)
```

sysadm_t는 sysctl파라메터를 변경할수 있다. 그것은 기초적으로 /proc/sys에 있는 모든것이다.

만일 지령

```
grep ^type. *sysctl_type policy.conf
```

를 실행하면 속성 sysctl_type를 가지고있는 형을 볼수 있다.

file_contexts파일

이 file_contexts파일은 보안방책이 설치될 때 체계에 파일을 적용하는 보안문맥을 포함한다.

이 파일은 setfiles프로그람에 의해서 읽어진다. 그리고 파일을 표식하는 정보를 사용한다.

```
# The security context for all files not otherwise specified.
/.*                                     system_u:object_r:file_t
```

이 행은 특수한 보안문맥을 가지고있지 않는 파일에 대한 보안문맥을 설정한다. system_u는 체계프로쎄스와 데몬(daemon)에 대한 신분이며 체계에 의해 속해있는 파일에 대한 기정신분이다.

The root directory.

/ -d system_u:object_r:root_t

실제적인 뿌리등록부(-d 가 기입됨)에 대한 root_t의 형을 가진 문맥을 설정한다. /mnt와 /initrd 역시 형 root_t를 가지고있다.

/home/[^/]+ -d system_u:object_r:user_home_dir_t

/home/[^/]+/.+ system_u:object_r:user_home_t

실지 /home등록부에 대해서 user_home_dir_t로 형을 설정한다. 그 아래에 있는 파일에 대해서는 user_home_t로 설정한다.

독자는 이 파일에서 그 밖의 모든것에 대하여 일반적인 리해를 할수 있다. 그리고 규칙적인 표현에 대한 리해를 가질수 있을것이다.

d는 등록부와 련관되여있다. 아무것도 목록화되여있지 않으면 정합되는것이 없다는 것을 의미한다. 만일 당신이 《ls -l》지령을 실행하면 출력문의 첫 항목에서 문자는 중간항목에서 나타나는것이다. 그래서 어떤것이 기호련결이였다면 블로크장치를 비롯해서 -l, -b를 보게 된다.

5) 형등록부(types directoriy)

이 등록부는 형정의를 포함하는데 다음의 파일들로 나누어진다.

device.te

이 파일은 장치말단의 형을 포함한다.

type device_t, file_type;

이 행은 /dev에 대한 형 device_t를 정의한다. file_type는 등록부와 파일에 모든 형에 대해서 사용되는 속성이다.

만일 파일 file_context에서 /dev를 탐색하면 그 형이 device_t로 설정되여있는것을 볼수 있다.

type null_device_t, file_type, device_type, mlstrustedobject;

/dev/null에 대해서는 형 null_device_t를 정의한다. device_t속성은 장치말단에 할당된 모든 형을 정의한다. Mls믿음오브젝트는 여기서 사용되지 않는다.

devpts.te

이 파일은 가상(pseudo)ttys에 대한 형을 포함한다.

type devpts_t, fs_type, root_dir_type;

devpts파일체계(devpts_t)의 형을 설정하고 그 파일체계의 뿌리등록부의 형을 설정한다.

file.te

파일의 형을 포함한다.

type unlabeled_t, sysadmfile;

표식화되지 않은 오브젝트는 형 unlabeled_t를 가지고있다. 만일 방책을 변경하여 형정의를 제거할 때에는 그 형을 사용하는 모든것이 표식화되지 않는다.

network.te

망의 형을 포함한다.

type netif_t, netif_type;

type netif_eth0_t, netif_type;

type netif_eth1_t, netif_type;

type netif_eth2_t, netif_type;

type netif_lo_t, netif_type;

type netif_ippp0_t, netif_type;

netif 형은 망대면부에 사용된다.

nfs.te

NFS사용에 대한 형을 포함한다.

type nfs_t, fs_type, root_dir_type;

nfs_t는 NFS파일체계와 그 파일들에 대한 기정형이다. NFS파일체계의 뿌리등록부를 형이 nfs_t가 되게 설정한다.

procfs.te

이 파일은 proc파일체계에 대한 형을 포함한다.

type proc_t, fs_type, root_dir_type;

type proc_kmsg_t;

type proc_kcore_t;

proc_t는 proc파일체계의 형이다. Proc_kmsg는 /proc/kmsg에 대한 형이다. proc_kcore_t는 /proc/kcore에 대한 형이다.

security.te

이 파일은 SELinux에서 보안 stuff(자료)에 대한 형을 포함한다.

type security_t, fs_type;

type policy_config_t, file_type;

type policy_src_t, file_type;

security_t는 보안클라스에서 권한을 검사할 때 목표형이다. policy_config_t는 /etc/security/selinux/*의 형이며 policy_src_t는

/etc/selinux/*에 대한 형이다.

6) 마크로등록부

우리는 이미 파일 user_macro.te와 admin_macros.te를 취급하였다. 마크로등록부에서 두개의 서로 다른 파일은 core_macros.te와 global_macros.te이다.

core_macro.te

이 파일은 자주 변경되지 않는 마크로를 포함하고있으며 그것들을 변경하지 않을것을 권고한다. 그것은 핵심부관련방책이 방책을 공유하려는것과 같기때문이다. 이 파일에서의 변경은 자신의 방책을 모든 사람이 다 사용하게 하는 불합리성을 가진다. 이 파일을 변경하였다면 이것은 독자가 다른 사람과 서로 다르게 작업하는 체계를 가지는것으로 될것이며 그것은 그리 흥미가 없다.

이 파일에 포함된 마크로의 일부는 클라스들과 허용권한들(permission)의 그룹화에 대한 마크로들이다.

define(`dir_file_class_set', `{ dir file lnk_file sock_file fifo_file chr_file blk_file }')

이 행은 마크로 dir_file_class를 정의한다. 이 마크로는 클라스 dir(등록부용), file(파일용), lnk_file(기호련결용), sock_file(Unix도메인소케트용), fifo_file(pipes로 이름지어진것), chr_file(문자블로크장치용), blk_file(블로크장치용)을 포함한다.

define(`rw_file_perms', `{ ioctl read getattr lock write append }')

이 파일은 허용권한 ioctl, read, getattr 그리고 lock, write, append를 포함하는 rw_file_perms마크로를 정의한다.

global_macros.te

이 마크로는 체계전반에 대한 마크로를 포함한다. 특수방책파일로 묶어져있는것을 의미하지 않는다.

```
define(`can_setexec',`
allow $1 self:process setexec;
allow $1 proc_t:dir search;
allow $1 proc_t:{ file lnk_file } read;
allow $1 self:dir search;
allow $1 self:file { read write };
')
```

마크로 can_setexec를 정의한다. $1은 실행문맥을 설정할수 있다. 그래서 그것은 자식프로쎄스의 문맥을 설정할수 있다.

$1은 /proc를 탐색하고 그 등록부에 있는 파일과 기호련결들을 읽을수 있다.

macros/program 등록부

이 program보조등록부는 사용자마다(per-user) 역할방책을 필요로 하는 프로그람에 대한 추가마크로를 포함한다. ssh와 같은 프로그람들은 사용자마다 역할방책을 요구

한다. 그것은 파생도메인이 사용자도메인호출에 기초하고있기때문이다.

만일 ssh_macro.te를 주시해보면 다음과 같은것을 볼수 있다.

define(`ssh_domain',`

Derived domain based on the calling user domain and the program.

type $1_ssh_t, domain, privlog;

만일 user_t가 호출사용자도메인이였다면 파생도메인은 user_ssh_t일것이다. 마찬가지로 staff_t가 호출사용자 도메인이라면 staff_ssh_t는 파생도메인일것이다.

행

domain_auto_trans($1_t, ssh_exec_t, $1_ssh_t)

는 호출도메인으로부터 파생도메인으로 전환을 허락한다.

7) flask등록부

다음과 같은 파일을 포함하고있다.

Access_vectors

이 파일은 다양한 클라스에 대해 집행할수 있는 동작을 정의한다. 파일클라스에 대해서 사용자는 읽기, 쓰기, 련결 등과 같은 동작을 진행한다. 소케트클라스에 대해서 결합(TCP와 UDP소케트와 같은 소케트를 결합), 접속호출(listen), 접속 기타 등등과 같은 동작들을 집행할수 있다.

initial_sids

이 파일은 초기의 SID(보안식별자)들을 정의한다. 오랜 SELinux에서 SIDS는 핵심부에 대한 리용자공간대면부에서 쓰이였다. PSID(완고한 SID)들은 디스크상의 파일과 등록부들에 대한 문맥들로 파일을 배치하는 핵심부코드에 리용되였다. 새로운 SELinux에서 확장된 속성들은 문맥을 포함하고있으므로 SID들과 PSID들은 더 이상 필요없다. 비록 새로운 SELinux가 확장속성을 리용하여도 일부초기문맥들은 여전히 체계가 시동할 때 정의되여야 한다. init_sids파일은 초기의 SID상수들을 포함한다. 보안방책원천등록부에서 파일 initial_sid문맥들은 이 초기SID들을 문맥들로 배치된다. 아래에 실례를 보여준다.

sid kernel   system_u:system_r:kernel_t

sid security   system_u:object_r:security_t

첫번째 행은 핵심부의 초기SID를 정의하며 system_u:system_r:kernel_t 의 문맥을 얻는다. kernel_t 은 일반적인 핵심부코드의 형이다. 두번째 행은security_t가 /selinux파일체계에 대한 형인 system_u:object_r:security_t의 문맥으로 sid security를 준다.

security_classes

이 파일은 보안객체클라스들을 정의한다. 이것들은 파일들과 망환경(networking)과 같은것들에 대한 클라스들이다.

8) 보안방책의 편집

보안방책을 편집하는 가장 좋은 방도는 그 모두를 정확히 실행시켜보는것이다. 먼저 다른 방책들이 /etc/selinux/domains/program/에 이미 씌여있으며 해당 file_contexts파일이 /etc/selinux/file_contexts/program/에 있는가를 확인한다.

아래에서는 보안방책을 편집하려고 하는 때에 알아두어야 할 몇가지 비결에 대하여 서술한다.

변화시키고 편집하려는것을 명백히 한다.

변화시키려고 하는것에 대한 리해를 잘하는것이 중요하다. 실례로 user_r역할에서 리용자들이 기정설정에서 허가되지 않은 어떤 등록부를 볼수 있도록 하려는것과 같은 허가하려는 동작이 있는가? domain/들이 무엇을 포함하는가? 어떤 마크로들을 호출하려고 하는가? 현존 규칙들을 보고 문법의 사상을 얻는다. 마크로등록부에 있는 마크로들을 찾아보고 그것들이 무엇을 하는가를 알아낸다.

전용화하는 규칙을 가진 파일을 생성한다.

보조등록부domains/misc/에 custom.te라고 하는 파일을 가지고있다.(자신의 보안방책원천등록부아래에) 이 파일에 독자가 하려고 하는것에 대하여 전용화된 자기 자신의 규칙들을 포함시킨다. 독자는 이 파일을 시험재료로 리용할수 있다.

핵심부통보문을 리해한다.

만일 원하지 않던 일이 일어나면 핵심부통보문을 검토한다. 보안방책을 쓰는데서 많은 품이 드는것은 일지기록들을 연구하고 그 다음 규칙들을 추가하거나 변경하여 일지기록들에 렬거된 오유들을 소거하는것이다. 실례로 staff_t가 tcpdump를 실행하려고 하는데 그 조작이 거부되면 일지기록을 검토한다. 그러면 다음과 같은것을 볼수 있다.

avc: denied { create } for pid=17824 exe=/usr/bin/traceroute.lbl scontext=faye:staff_r:staff_t tcontext=faye

:staff_r:staff_t tclass=rawip_socket

여기서 우리는 staff_t도메인에서 누군가가 traceroute지령을 실행하려고 하였다가 거부되였다는것을 알수 있다. 이 일지통보문으로부터 우리는 traceroute가 도메인 staff_t로 실행하고있었다는것을 알수 있지만 그것이 경로추적(traceroute)할수 있는 특권도메인으로 실행되도록 하려고 한다. 우리가 하려고 하는것은 우리의 방책을 편집하여 그것이 형 traceroute_exec_t(경로추적프로쎄스용도메인)를 실행할 때 staff_t가 traceroute_t(이것은 우리가 traceroute지령을 실행한 후 동작하는 traceroute프로쎄스와는 반대로 실제적인 경로추적실행의 형이다.)로 이행하는것을 허용하는것이다.

9) 기본적인 보안방책편집실례

아래에서 보여주는것은 보안방책을 편집하는데서 도움이 되는 실례들이다. 보안방책파일을 보관한 후 방책원천등록부에서 make load를 실행하여야 한다는것을 명심하시오.

user_t가 tcpdump를 리용하도록 허가한다.

이미 전용화파일(이것을 custom.te이라고 하자.)생성하지 않았으면 방책원천등록부 아래의 보조등록부domains/misc/에 그것을 생성한다. 다음의 행을 추가한다.

```
# your own comment here
domain_auto_trans(userdomain, netutils_exec_t, netutils_t)
in_user_role(netutils_t)
allow netutils_t user:chr_file rw_file_perms;
```

무엇보다 먼저 우리는 리용자도메인(여기서 userdomain은 가능한 모든 리용자도메인 즉 user_t, staff_t, sysadm_t 그리고 사용자가 가질수 있는 기타 모든 리용자도메인들을 참조한다.)으로부터 실제적인 tcpdump프로쎄스에 대한 도메인(이것이 netutils_exec_t 이다.)으로 이행할수 있게 하려고 한다. 첫번째 행은 리용자도메인이 tcpdump실행을 진행할 때 자동적인 이행이 netutils_t인 tcpdump프로쎄스도메인으로 진행된다는것을 보여준다

in_user_role 마크로(파일user_macros.te에서 정의된)는 파라메터(이 경우 netutils_t)로서 넘겨진 도메인이 모든 사용자역할들(user_r 와 staff_r와 같이. sysadm_r은 관리자적인(administrative)역할이며 리용자역할은 아니다.)로 되는것을 허가한다. 이 행은 역할 user_r와 도메인 netutils_t의 어떤 결합이 보안문맥에서 유효하도록 하는데 필요하다.

세번째 행은 도메인 netutils_t가 리용자 pty형들을 호출하도록 한다. netutils_t는 이 호출을 통하여 사용자가 자기의 말단장치로부터 읽거나 말단장치에 쓸수 있게 한다. chr_file은 말단장치에 쓰기하는데 쓰인다.

련습 1: 이 행들을 가지고 진행한다. allow행을 주석처리하고 방책을 다시 적재하고 tcpdump를 시도한다. 일지기록을 통하여 아무것도 일어나지 않은것처럼 보이는 원인에 대하여 검토한다.

련습 2: allow행을 주석처리한채로 가상조작탁(이전에 xterm을 리용하고있다고 가정하고)으로 절환하고 거기로부터 tcpdump를 시도한다. 만일 명확히 tty장치로부터 tcpdump접근을 허가하지 않았다면 그것이 일어나도록 시도한다. pty장치로부터 tcpdump를 할수 있지만 tty장치로부터는 할수 없다.

user_t가 /etc/selinux/ 등록부를 읽도록 허가한다.

보통 독자가 비특권리용자들이 /etc/selinux/에 무엇이 있는지 아는것을 원하지 않지만 배우기 위한 목적으로 여기서는 이 실례를 고찰한다. 독자는 custom.te파일을

편집하여 다음의 내용을 추가한다.

your comment here

r_dir_file(user_t,policy_src_t)

r_dir_file파일은 그 바로 아래에 있는 등록부와 파일들을 읽는것을 허가한다. user_t 는 도메인이며 policy_src_t는 /etc/selinux의 형이다.

련습 1: custom.te 을 편집하기 전과 후에 /etc/selinux를 호출하여본다.(그리고 방책을 다시 적재한다.) 무엇이 일어나는가를 보기 위해 일지기록들을 검토한다.

련습 2: user_t 로부터 /boot를 호출하여 본다. 《permission denied》를 얻는가? user_t 가 이 등록부를 읽는것을 허가하는 규칙을 생성한다.

새로운 형을 창조하기

이 실례에서 우리는 custom.te에 다음과 같은 행을 추가하여 우리자신의 새로운 파일형을 창조하고 그 다음 make load지령을 실행한다.

```
type ourtype_t,file_type,sysadmfile;
allow staff_t ourtype_t:file { create_file_perms relabelfrom relabelto };
```

우리는 《ourtype_t》로 불리우는 새로운 형을 정의하고 그것을 속성 file_type와 sysadmfile로 하여 관리자가 그것을 호출할수 있도록 한다. 두번째 행은 staff_t가 형 ourtype_t인 파일들에 완전한 접근을 가진다는것을 보여준다.(읽기와 쓰기 등) relabelfrom와 relabelto는 staff_t 가 또 다른 형으로부터 그리고 또 다른 형으로 형 ourtype_t인 파일들을 재표식할수 있다는것을 의미한다.

이제 staff_t역할로 새 파일을 생성한다. 다음과 같은 그 파일의 보안문맥을 검토하시오.

```
faye@kaos:~$ ls -Z foo
-rw-r--r--  faye      faye      faye:object_r:staff_home_t        foo
```

So we see file "foo" has the type staff_home_t. Now change that type to ourtype_t:

```
faye@kaos:~$ chcon -t ourtype_t foo
faye@kaos:~$ ls -Z foo
-rw-r--r--  faye      faye      faye:object_r:ourtype_t           foo
```

이제 우리가 방금 생성한 형을 제거하자. 다시 한번 custom.te파일을 편집하여 방

금 추가한것을 주석처리한다. 다시 make load를 실행하고 파일의 속성을 본다.

faye@kaos:~$ ls -Z foo

ls: foo: Permission denied

sysadm_r가 ls와 같은 지령을 실행하면 다음과 같다.

-rw-r--r-- faye faye faye:object_r:ourtype_t /home/faye/foo

이제 staff_r로서 파일 foo에 접근하려고 할 때 일지에 기록된 오유에 대한 일지들을 검토하자.

avc: denied { getattr } for pid=29494 exe=/bin/ls path=/home/faye/foo dev=md7 ino=145445 scontext=faye:staff

_r:staff_t tcontext=system_u:object_r:unlabeled_t tclass=file

목표문맥은 파일 foo에 대하여 형unlabeled_t를 포함한다는것을 명심하시오. 우리가 방책으로부터 형 ourtype_t를 제거하였다면 우리가 그 형으로 생성하였던 파일들이 형 unlabeled_t로 재표식된다. 비록 ls Z가 형이 ourtype_t 라고 보여주어도 핵심부는 ourtype_t가 방책에 존재하지 않으면 그것을 unlabeled_t로 간주한다.

제 8 장. 체계관리

제 1 절. 핵심부동기화

핵심부는 어떤 요구에 응답하는 일종의 봉사기로 생각할수 있다. 핵심부에 대한 요구는 CPU에서 실행중인 프로쎄스가 할수도 있고 새치기요구를 발생시키는 외부장치가 할수도 있다. 따라서 핵심부에서는 경쟁상태(race condition)가 발생할수 있으며 적당한 동기화기법으로 이것을 조종해야 한다.

우선 어느 때 얼마나 많은 핵심부요구가 동시에 발생하는가에 대하여 설명한다. 그 다음 핵심부에서 실현하는 기본적인 동기화기법을 소개하고 대부분의 일반적인 상태에서 이것을 적용하는 방법을 설명한다. 마지막으로 몇가지 실례들을 설명한다.

1. 핵심부조종경로

앞에서 핵심부조종경로를 핵심부가 서로 다른 종류의 새치기를 처리하기 위해 실행하는 일련의 명령어라고 정의하였다. 매개의 핵심부에 대한 요구를 서로 다른 핵심부조종경로로 실행하고 매 핵심부조종경로는 보통 몇개의 핵심부함수를 실행한다. 례를 들어 사용자방식프로쎄스가 체계호출을 요구하면 이에 해당한 핵심부조종경로는 system_call()함수에서 시작해서 ret_from_sys_call()함수(《새치기와 례외에서 되돌이》참고)에서 끝난다.

앞에서 설명한것처럼 여러가지 방법으로 핵심부요구가 발생한다.

・사용자방식에서 실행중인 프로쎄스가 례외를 발생시킨다. 례를들면 int 0x80기호언어명령어를 실행하는 경우.

・외부장치가 IRQ선을 통해 프로그람가능한 새치기조종기 (PIC:programmable interrupt controller)에 새치기신호를 보낼 때 해당 새치기를 허용하고있는 경우.

・핵심부방식에서 실행중인 프로쎄스가 페지절환(page fault)례외를 일으키는 경우.(《페지오유례외조종기》참고)

・다중처리기체계에서 핵심부방식에서 동작중인 프로쎄스가 처리기들사이에 새치기를 발생시키는 경우. (《처리기간의 새치기처리》참고)

핵심부조종경로는 프로쎄스와 비슷한 역할을 수행하지만 이것들이 매우 기초적인것이다.

첫째로, 핵심부조종경로에는 서술자가 련결되지 않는다.

둘째로, 핵심부조종경로를 실행하도록 순서짜기하는것은 한개의 함수로 이루어지지 않고 핵심부코드중간에 들어있는 핵심부조종경로를 정지하거나 복귀하게 하는 일련의 명령어를 통해 이루어진다.

제일 간단하게는 CPU가 핵심부조종경로를 첫 명령어부터 마지막명령어까지 차례로 실행한다. 그렇지만 아래의 사건들가운데서 한가지 사건이라도 발생하면 CPU는 다른

핵심부조종경로를 끼워넣는다.

・프로쎄스절환(process switch)이 발생한 경우. schedule()함수를 호출할 때에만 프로쎄스절환이 발생한다.

・새치기를 허용한 상태에서 CPU가 핵심부조종경로를 실행하고있을 때 새치기가 발생한 경우. 이 경우 이전핵심부조종경로를 끝나지 않은 상태로 둔채로 CPU는 새치기를 처리하는 다른 핵심부조종경로를 수행하기 시작한다.

・지연함수(deferrable function)를 실행하는 경우. 이미 앞에서 설명한것처럼 새치기발생이나 local_bh_enable()함수를 호출하는것과 같은 여러가지 사건에 의해 지연함수를 실행하게 된다.

다중처리를 실현하려면 핵심부조종경로를 동시에 실행하는것이 중요하다. 《레외조종기와 새치기조종기의 동시실행》에서 설명한것처럼 동시실행은 프로그람가능한 새치기조종기와 장치조종기의 효률을 높인다. 핵심부조종경로를 동시에 실행할 때에는 완충기와 그 길이를 나타내는 정수형변수같은 서로 련관된 성원변수를 포함하는 자료구조체를 사용하는 경우에 특히 주의해야 한다. 이런 자료구조체에 영향을 미치는 모든 코드는 제한된 령역(critical region)에 들어가야 한다. 그렇지 않을 경우 자료구조체가 혼동이 될수도 있다.

앞에서 설명한것처럼 Linux핵심부는 비선취형이다. 즉 프로쎄스가 핵심부방식에서 실행중인 동안에는 선취(우선순위가 더 높은 프로쎄스로 교체하는것)할수 없다.

다음표는 선취조종을 위한 prempt_count의 성원마크로들이다.

마크로	기능
preempt_count( )	tHRead_info서술자에서 preempt_count를 선택한다.
preempt_disable( )	선취계수기의 값을 하나 증가한다.
preempt_enable_no_resched( )	선취계수기의 값을 하나 감소한다.
preempt_enable( )	선취계수기값을 하나 증가하고 thread_info서술자에서 TIF_NEED_RESCHED기발이 설정되였다면 preempt_schedule( ) 함수를 호출한다.

get_cpu( )	preempt_disable( )과 비슷하지만 국부적인 cpu개수를 되돌린다.
put_cpu( )	preempt_enable( )과 같다.
put_cpu_no_resched( )	preempt_enable_no_resched( )와 같다.

특히 Linux는 다음의 원칙을 항상 준수한다.

・핵심부방식에서 실행중인 프로쎄스가 자발적으로 CPU에 대한 조종권을 되돌리지 않는 이상 이것을 다른 프로쎄스로 교체하지 않는다.

・새치기나 례외, 프로그람적인 새치기를 처리하려고 핵심부방식에서 실행중인 프로쎄스를 중단할수는 있지만 조종기를 마치면 해당 프로쎄스의 핵심부조종경로로 복귀한다.

・지연함수나 체계호출봉사루틴을 실행하는 핵심부조종경로를 실행하려고 새치기를 처리하는 핵심부조종경로를 멈출수 없다.

때문에 단일처리기체계에서 차단(block)을 일으키지 않는 체계호출을 처리하는 핵심부조종경로는 체계호출에 의해 시작한 다른 핵심부조종경로를 그대로 리용한다.

이것은 많은 핵심부함수를 쉽게 실현할수 있게 한다. 새치기나 례외, 프로그람적인 새치기조종기가 갱신하지 않는 모든 핵심부자료구조체에 안전하게 호출할수 있다.

그러나 핵심부방식에서 동작중인 프로쎄스가 자발적으로 CPU에 대한 조종권을 되돌린다면 모든 자료구조체를 정확한 상태로 보존해야 한다. 또한 실행을 다시 시작할 때에는 이전에 사용한 자료구조체의 값이 수정되였을수도 있으므로 그 값을 다시 검사해야 한다. 자료구조체는 서로 다른 프로쎄스의 문맥에서 실행한 다른 핵심부조종경로에 의해 (때로는 같은 코드를 실행했을수도 있다.)수정될수 있다.

다중처리기체계에서는 모든것이 훨씬 더 복잡하다. 많은 CPU가 동시에 핵심부코드를 실행할수 있기때문에 핵심부개발자는 어떤 자료구조체를 새치기나 례외, 프로그람적인 새치기조종기에서 절대로 변경하지 않아도 이것을 안전하게 호출할수 있다고 가정할수 없다.

이 장의 나머지 부분에서 동기화가 필요할 때 무슨 일을 해야 하는가, 즉 공유하는 자료구조체를 호출하다가 자료를 파괴하는것을 어떻게 막을것인가를 설명한다.

2. 동기화기법

앞에서 프로쎄스에서의 경쟁상태와 제한령역을 소개하였다. 이와 똑같은 정의를 핵심부조종경로에 대해서도 할수 있다. 핵심부에서는 둘이상의 핵심부조종경로가 어떻게 겹치는가에 따라 계산결과가 다르게 나올 때 경쟁상태가 발생한다.

《제한령역(critical region)》은 핵심부조종경로가 시작되면 다른 핵심부조종경로가

시작하기 전에 완전히 수행을 마쳐야 하는 모든 코드령역을 말한다. 이제 공유자료사이에서 일어날수 있는 경쟁상태를 피하면서 핵심부조종경로를 동시에 실행하는 방법을 보자. 표 5-1에 Linux핵심부에서 사용하는 동기화기법을 표시하였다. 표에서 《범위》렬은 그 동기화기법이 체계에 있는 모든 CPU에 적용되는가 아니면 한 CPU에만 적용되는가를 나타낸다. 례를 들어 국부새치기를 금지하는것은 한 CPU에만 적용된다.(체계에 있는 다른 CPU는 영향을 받지 않는다.) 반대로 원자적인 연산은 체계에 있는 모든 CPU에 적용된다.(원자적인 연산을 사용하면 여러 CPU가 같은 자료구조체에 호출하더라도 이것이 겹치지 않는다.)

표 8-1. 핵심부가 사용하는 다양한 종류의 동기화기법

기 법	설 명	범 위
개별적CPU변수	CPU들에 대한 자료구조체를 중복	모든 CPU
원자적인 연산	계수기(counter)에 원자적인 읽기/수정/쓰기명령	모든 CPU
기억기장벽	명령어를 재배치하는것을 막기	국부 CPU
스핀잠그기	바쁘게 기다리는 잠그기	모든 CPU
신호기	대기(잠자기)를 차단하며 잠그기	모든CPU
국부새치기금지	한 CPU의 새치기처리를 금지	국부CPU
국부프로그람적새치기금지	한 CPU의 지연함수처리금지	국부CPU
읽기-복사-갱신(RCU)	지시자를 통해서 공유된 자료구조체에 대한 접근을 잠그기해제	모든 CPU

이제 매개의 동기화기법을 간단히 보자. 나중에 《핵심부자료구조체로의 호출동기화》에서 핵심부자료구조체를 보호하기 위해 이런 동기화기법을 어떻게 결합하는가를 설명한다.

1) 개별적CPU변수리용

첫 단계에서 동기화에 대한 필요성을 피하지 않는한 핵심부설계에서 제일 좋은 동기화기법으로 되고있다. 앞으로 보겠지만 실제로 모든 동기화기법들은 수행률이 중요하다. 개별적인 CPU변수들을 핵심부변수로 선언함으로써 가장 간단하고 그중 효과적인 동기화기법으로 된다. 기초적으로 매 CPU변수는 체계에서 하나의 CPU에 해당한 하나의 원소인 자료구조체배렬로 된다.

하나의 CPU는 다른 CPU들에 따르는 배렬원소들에 접근할수 없어야 한다. 다시 말해서 경쟁상태에 대한 고려가 없이 자기의 원소를 자유롭게 읽고 변경시킬수 있게 한다는것이다. 해당 CPU에게만 그렇게 할 자격을 주어야 하기때문이다.

이것은 또한 체계의 CPU들에 대한 자료를 론리적으로 갈라보면 개별적인 CPU변수들을 기본적으로 특별한 경우에만 리용할수 있다는것을 의미한다. 매 CPU배렬의 원소들은 주기억기에 차례로 들어가 매 자료구조체가 하드웨어캐쉬의 서로 다른 흐름상에 놓이도록 한다.

그러므로 개별적인 CPU배렬에 대한 동시접근은 캐쉬흐름조사와 무효화의 결과를 낳는다. 이것은 체계동작에서 품이 드는 조작이라고 볼수 있다. 개별적인 CPU변수들은 여러개의 CPU들로부터의 동시접근에 대해 보호를 제공하지만 비동기적인 함수들(새치기처리기와 지연함수들)로부터의 접근에 대한 보호는 제공하지 않는다. 이 경우에 추가적인 동기화기법들이 요구된다.

더우기 개별적인 CPU변수들은 단일처리소자체계와 다중처리소자체계에서 다 핵심부선취권에 의해 경쟁상태에 자주 부닥치게 된다. 일반적인 규칙으로 보면 핵심부조종경로는 핵심부선취를 불가능하게 하고 개별적인 CPU변수들에 접근해야 한다. 구체적으로 본다면 핵심부조종경로가 매 CPU변수들의 국부복사주소를 얻을 때 무슨 일이 일어나는가를 정확히 보고 다음 다른 CPU에 선취하여 이동할 때 주소는 이전 CPU의 원소를 계속 참조한다.

표 8-2은 개별적CPU들에 대한 가변성을 리용하기 위해 핵심부에서 제공하는 주요 함수들과 마크로목록이다.

표 8-2. 개별적인 CPU변수들에 대한 함수와 마크로들

마크로, 함수이름	설 명
DEFINE_PER_CPU(type, name)	표준자료구조체의 이름으로 불리우는 개별적인 CPU 배렬을 정적으로 할당한다.
Per_cpu(name, cpu)	CPU원소를 개별적인 CPU배렬이름으로 선택한다.
_ _get_cpu_var(name)	매 CPU배렬이름중 국부 CPU원소를 선택한다.
Get_cpu_var(name)	핵심부선취를 불가능하게 하고 매 CPU배렬이름중 국부 CPU의 원소를 선택한다.
Put_cpu_var(name)	핵심부선취가능(이름은 리용안됨)
alloc_percpu(type)	표준자료구조체의 개별적CPU배렬을 동적으로 할당하고 그의 주소를 되돌린다.
Free_percpu(pointer)	주소지시자에 동적으로 할당된 개별CPU배렬을 해제

	한다.
Per_cpu_ptr(pointer, cpu)	주소지시자에 개별CPU배렬중 CPU원소의 주소를 되돌린다.

2) **원자적인 연산**

여러가지 기호언어명령어는 《읽기/수정/쓰기》류형이다. 즉 기억기주소를 두번 호출하면서 처음에는 이전값을 읽고 다음에는 새로운 값을 쓴다. 두 CPU에서 두개의 핵심부조종경로가 원자적이지 않은 연산을 사용하여 동시에 같은 기억기주소에 《읽기/수정/쓰기》를 실행한다고 하자. 먼저 두 CPU는 똑같은 기억기주소를 읽으려고 시도하면 《기억기중재자(memory arbiter, 주기억기소편에 대한 호출을 직렬화하는 하드웨어회로)》가 이것들가운데서 하나에만 호출을 허가하고 다른 CPU의 호출을 지연한다. 그러다가 첫번째 읽기동작이 끝나면 지연된 CPU는 같은 값(이전값)을 해당 기억기주소에서 읽는다. 다음으로 두 CPU가 모두 똑같은 값(새로운 값)을 해당 기억기주소에 쓴다. 다시 기억기중재자가 나서서 모선기억기로의 호출을 직렬화하고 마침내 두번의 쓰기작업이 모두 성공한다.

그런데 두개의 CPU가 모두 똑같은 값(새로운 값)을 기록했으므로 결과는 잘못된것이다.

그 결과 중첩된 두 《읽기/수정/쓰기》연산은 마치 하나인것처럼 된다.

《읽기/수정/쓰기》명령어에 의한 경쟁상태를 막을수 있는 가장 쉬운 방법은 이런 연산을 소편(chip)준위에서 원자적으로 처리하는것이다. 이런 연산은 중간에 방해받거나 다른 CPU가 똑같은 기억기주소에 호출하지 못하게 만들어 한개의 명령어로 실행해야 한다. 제한령역을 만드는 좀 더 유연한 다른 기구의 기반으로 이런 매우 적은 수의 원자적인 연산을 사용하는것을 볼수 있다. (atomic.h)

이제 이 분류에 따라 Intel 80x86 명령어를 살펴보자.

・기억기를 호출하지 않거나 정렬된 호출을 한번 하는 기호언어명령어는 원자적이다.

・inc나 dec같이 기억기에서 자료를 읽고 갱신하고 기억기에 다시 쓰는 읽기/수정/쓰기 기호언어명령어는 읽기와 쓰기사이에 다른 처리기가 기억기모선를 점유하지 않는 한 원자적이다. 단일처리기(uniprocessor)체계에서는 기억기모선를 가로채는 일이 발생하지 않는다.

・연산코드앞에 lock바이트(0xf0)앞붙이가 붙은 읽기/수정/쓰기기호언어명령어는 다중처리기체계에서도 원자적이다. 조종장치는 이것을 앞붙이를 만나면 명령어수행을 끝마칠 때까지 기억기모선을 잠그기한다. 따라서 잠그기가 걸린 명령어를 실행하는 동안에는 기억기를 다른 처리기가 호출할수 없다.

·조종장치가 같은 명령을 여러번 반복할수 있도록 하는 rep바이트 (0xf2, 0xf3) 앞붙이가 연산코드앞에 붙은 기호언어명령어는 원자적이지 않다.

조종장치는 다음 반복을 시작하기 전에 대기중인 새치기가 있는가를 검사한다.

C코드를 작성할 때 a=a+1이나 또는 a++와 같은 연산을 사용해도 콤파일러가 이것들을 원자적인 명령어를 사용하도록 코드를 생성한다고 장담할수 없다. 따라서 Linux 핵심부는 특별히 atomic_t형(원자적으로 호출할수 있는 24bit계수기)과 actomic_t변수를 가지고 동작하고 원자적인 한개의 기호언어명령어로 실행되는 특별한 함수를 제공한다.(표 8-3 참고) 다중처리기체계에서는 매 명령어앞에 lock앞붙이가 붙는다.

표 8-3. Linux에서의 원자적인 연산

함 수	설 명
Atomic_read(v)	*v를 되돌린다
Atomic_set(v,i)	*v를 i로 설정한다
Atomic_add(I,v)	*v에 i를 더한다
Atomic_sub(I,v)	*v에서 i를 던다
Atomic_sub_and_test(i,v)	*v에서 i를 던 후 결과가 0이면 1, 0이 아니면 0을 되돌린다
Atomic_inc(v)	*v에 1을 더한다.
Atomic_dec(v)	*v에서 1을 던다.
Atomic_dec_and_test(v)	*v에서 1을 던 후 결과가 0이면 1, 0이 아니면 0을 되돌린다.
Atomic_inc_and_test(v)	*v에 1을 더한 후 결과가 0이면 1, 0이 아니면 0을 되돌린다.
Atomic_add_negative(i, v)	*v에 i를 더한 후 결과가 부수이면 1, 부수가 아니면 0을 되돌린다
atomic_inc_return(v)	*v에 1을 더하고 *v의 새값을 되돌린다.
atomic_dec_return(v)	*v에서 1을 덜고 *v의 새값을 되돌린다.
atomic_add_return(i, v)	*v에 I를 더하고 *v의 새값을 되돌린다.
atomic_sub_return(i, v)	*v에서 I를 덜고 *v의 새값을 되돌린다.

비트마스크(bit mask)에 동작하는 다른 부류의 원자적인 함수도 있다.(표 8-4 참고) 이런 경우 비트마스크는 일반정수변수이다.

표 8-4. Linux에서 비트를 다루는 원자적인 함수

함 수	설 명
test_bit(nr,addr)	*addr의 n번째 비트값을 되돌린다.
set_bit(nr,addr)	*addr의 n번째 비트를 1로 설정한다.
clear_bit(nr,addr)	*addr의 n번째 비트를 0으로 지운다.
change_bit(nr,addr)	*addr의n번째 비트값을 반전시킨다.
test_and_set_bit(nr,addr)	*addr의 n번째 비트를 1로 설정하고 이전값을 되돌린다.
test_and_clear_bit(nr,addr)	*addr의 n번째 비트를 0으로 지우고 이전값을 되돌린다.
test_and_change_bit(nr,addr)	*addr의 n번째 비트값을 반전시키고 이전값을 되돌린다.
atomic_clear_mask(mask,*addr)	addr에서 mask로 지정한 비트를 모두 0으로 지운다.
atomic_set_(mask,*addr)	addr에서 mask로 지정한 비트를 모두 1로 설정한다.

3) 기억기장벽

최적화를 하는 콤파일러를 사용할 때에는 원천코드에 나와있는 순서대로 명령어가 실행되지 않는다. 례를 들어 콤파일러는 등록기를 사용하는 방법을 최적화하는 방식으로 기호언어명령어의 순서를 바꿀수 있다. 나가서 최근의 CPU는 대부분이 여러개의 명령어를 병렬로 실행하여 기억기호출순서를 바꿀수도 있다. 이런 식의 재배치를 통하여 프로그람의 실행속도를 크게 향상시킬수 있다. 그렇지만 동기화를 다룰 때에는 명령어를 재배치하는 일을 피해야 한다.

사실 모든 동기화함수는 기억기장벽역할을 한다. 동기화함수의 다음번에 있는 명령어를 동기화함수보다 먼저 실행한다면 뜻밖의 결과를 초래할수 있다. 《기억기장벽(memory barrier)》기초함수는 그 이후에 있는 작업을 시작하기 전에 이전에 있는 작업을 끝마치게 한다. 따라서 기억기장벽은 어떤 기호언어명령어도 통과할수 없는 방화벽과 비슷하다.

80x86처리기에서 다음과 같은 종류의 기호언어명령어는 기억기장벽의 역할을 하기때문에 《직렬화》한다고 말한다.

· 입출력포구에 동작하는 모든 명령어

· lock앞붙이가 붙은 모든 명령어(《원자적인 연산》참고)

· 조종등록기나 체계등록기, 오유수정등록기에 쓰기를 하는 모든 명령어(례를 들어 eflags등록기의 IF기발의 상태를 바꾸는 cli와 sti명령어)

· 몇가지 특별한 기호언어명령어

이 가운데서는 새치기나 례외조종기를 끝마치는 iret명령어가 있다.

Linux는 표 8-5에 있는 기억기장벽을 의한 기초함수 6개를 사용한다.

《읽기용기억기장벽》은 기억기에서 읽기를 수행하는 명령어에만 동작하고 《쓰기용기억기장벽》은 기억기에 쓰기를 수행하는 명령어에만 동작한다.

기억기장벽은 다중처리기체계와 단일처리기체계에서 모두 사용할수 있다. 다중처리기체계에서만 발생할수 있는 경쟁상태를 막으려고 기억기장벽을 사용할 때에는 smp_xxx() 기초함수를 사용한다. 이 함수는 단일처리기체계에서는 아무 일도 하지 않는다.

다른 기억기장벽은 단일처리기와 다중처리기체계에서 경쟁상태를 막기 위하여 사용한다.

표 8-5. Linux에서의 기억기장벽

마크로	설 명
mb()	다중처리기(MP)와 단일처리기(UP)에서 사용할수 있는 기억기장벽
rmb()	MP와 UP에서 사용할수 있는 읽기용기억기장벽
wmb()	MP와 UP에서 사용할수 있는 쓰기용기억기장벽
smp_mb()	MP에서만 사용할수 있는 기억기장벽
smp_rmb()	MP에서만 사용할수 있는 읽기용기억기장벽
smp_wmb()	MP에서만 사용할수 있는 쓰기용 기억기장벽

기억기장벽의 실현은 체계구조마다 다르다. 례를 들어 Intel환경에서 rmb()마크로는

asm volatile(“lock; addl $0,0(%%esp)” ::: ” memory”)

로 바뀐다.

asm명령어는 콤파일러에 뒤에 나오는 기호언어명령어를 삽입하라고 지시한다.

volatile은 콤파일러가 asm명령어를 프로그람에 있는 다른 명령어와 섞는것을 금지한다.

memory는 콤파일러가 이 기호언어명령어로 인하여 주기억기의 모든 주소에 있는 값이 바뀌였다고 가정하게 한다. 따라서 콤파일러는 코드를 최적화하기 위해 asm명령어가 나오기 전에 CPU등록기에 보관한 기억기주소의 값을 사용할수 없다. 마지막으로 lock; addl $0,0(%%esp) 기호언어명령어는 탄창의 맨우에 있는 기억기주소에 0을 추가한다. 이 명령어자체는 쓸모가 없지만 앞에 붙은 lock앞붙이는 이 명령어를 CPU의 기억기장벽으로 만든다. Intel에서 wmb()마크로는 이보다 간단하여 asm

volatile(“” :::” memory”)로 바뀐다. Intel처리기는 기억기쓰기호출의 순서를 절대로 바꾸지 않기때문에 코드에 직렬화를 하는 기호언어명령어를 삽일할 필요가 없다. 그렇지만 이 마크로는 콤파일러가 명령어를 섞는것을 금지한다. 다중처리기체계에서 앞에서 《원자적인 연산》에서 설명한 모든 원자적인 연산은 앞에 lock앞붙이가 붙어있기때문에 기억기장벽역할을 한다.

4) 스핀잠그기

《잠그기(locking)》는 광범하게 사용하는 동기화기법이다. 핵심부조종경로에서 공유자료구조체를 호출해야 하거나 제한령역에 들어가야 할 때에는 반드시 이에 대한 《잠그기(lock)》를 획득해야 한다. 잠그기기구를 통해 보호하는 자원은 문을 잠근 방안에 갇힌 자원과 매우 류사하다. 어떤 핵심부조종경로가 자원을 호출하려고 할 때에는 자물쇠(잠그기)를 획득하여 《문을 열려고》한다. 이것은 자원을 사용할수 있을 때에만 성공한다.

자원을 사용하고있는 동안에는 문은 계속 잠겨있다. 핵심부조종경로에서 잠그기를 해제하면 문에 걸린 자물쇠가 열려 다른 핵심부조종경로가 그 안에 들어갈수 있다.

그림 8-1은 잠그기사용을 보여준다. 여기서는 핵심부조종경로 5개(P0,P1,P2,P3,P4)가 제한령역 2개(C1,C2)에 호출하려고 한다. 핵심부조종경로 P0은 C1령역안에 있고, P2와 P4는 여기에 들어가려고 기다리고있다. 동시에 P1은 C2령역안에 있고 P3이 들어가려고 기다리고있다. P0과 P1을 동시에 실행할수 있다. 제한령역C3에 들어가려는 핵심부조종경로가 없기때문에 이에 대한 잠그기는 열린 상태이다.

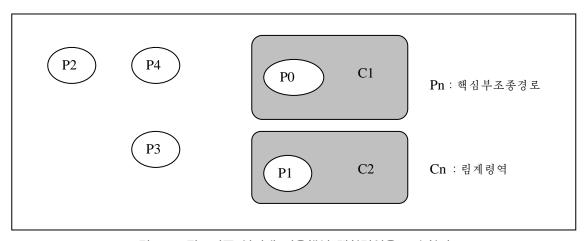



그림 8-1. 잠그기를 여러개 리용해서 제한령역을 보호하기

《스핀잠그기(spin lock)》는 다중처리환경에서 동작하도록 만든 특별한 종류의 잠그기이다. 핵심부조종경로는 스핀잠그기가 《열려》있으면 잠그기를 획득하여 실행을 계속하고 반대로 다른 CPU에서 실행중인 핵심부조종경로에 의해 스핀잠그기가 《잠겨》있으면 잠그기가 해제될 때까지 명령순환을 실행한다.

스핀잠그기를 단일처리기환경에서 사용하면 잠그기를 기다리는 핵심부조종경로는 실행을 계속하고 따라서 잠그기를 점유하는 핵심부조종경로는 잠그기를 해제할 기회가 없기때문에 단일처리기환경에서는 쓸모가 없다. 스핀잠그기가 실행하는 명령순환은 《바쁘게 기다림(busy wait)》이다. 스핀잠그기를 기다리는 핵심부조종경로는 시간을 랑비하는것외에는 할 일이 없어도 CPU에서 실행을 계속한다. 그럼에도 불구하고 많은 핵심부조종잠그기가 잠겨있는 시간은 1ms의 몇분의 1밖에 되지 않아 CPU를 해제하였다가 나중에 CPU를 다시 획득하는것이 오히려 시간을 더 소비하기때문에 일반적으로 스핀잠그기가 훨씬 편리하다.

Linux에서는 매 스핀잠그기를 한개의 lock마당으로 구성된 spinlock_t구조체로 나타낸다.

lock마당값은 열린 상태에서는 1이고 잠긴 상태에서는 0이거나 부수이다.

표 8-6은 스핀잠그기를 초기화하고 검사하고 설정할 때 사용하는 함수를 보여준다. 단일처리기체계에서는 항상 1을 되돌리는 spin_trylock()를 제외한 나머지 함수는 아무일도 하지 않는다. 아래에 있는 모든 함수는 원자적인 연산을 기반으로 한다. 이것은 CPU에서 실행중인 프로쎄스가 스핀잠그기를 갱신할 때 다른 CPU에서 실행중인 다른 프로쎄스가 동시에 그 스핀잠그기를 수정하려고 해도 스핀잠그기가 정확히 갱신되도록 보장한다. (\kernel\spinlock.c)

표 8-6. 스핀잠그기함수

마 크 로	설 명
Spin_lock_init()	스핀잠그기를 1로 설정한다.(열기)
Spin_lock()	스핀잠그기가 1로 될 때까지 (열기)기다렸다가 0으로 (잠그기) 설정한다.
Spin_unlock()	스핀잠그기를 1로 설정한다.(열기)
spin_unlock_wait()	스핀잠그기가 1로 될 때까지(열기) 기다린다.
Spin_is_locked()	스핀잠그기가 1이면 (열기)0, 아니면 1을 되돌린다.
Spin_trylock()	스핀잠그기를 0으로 설정한 후(잠그기) 잠그기를 획득한 경우 1, 실패한 경우 0을 되돌린다.

스핀잠그기를 획득할 때 사용하는 spin_lock마크로를 구체적으로 보면 다음과 같다.

이 마크로는 스핀잠그기의 주소 slp를 파라메터로 받아 다음과 같은 기호언어코드를 만들어낸다.

```
1: lock; decb slp
```

```
    jns 3f
2: cmpb $0,slp
    pause
    jle 2b
    jmp1b
3:
```

decb기호언어명령어는 스핀잠그기의 값을 감소시킨다. 명령어앞에 lock앞붙이가 붙어있으므로 이 명령어는 원자적이다.

다음으로 부호기발(sign flag)에 대해 검사를 수행하는데 그 값이 0이면 스핀잠그기가 1(열림)이라는것을 의미하므로 표식3에서 (표식뒤에 붙은 뒤붙이《:》은 표식이 현재명령어《뒤》에 있음을 의미한다. 이것은 나중에 나올 프로그람에서도 동작한다.)정상적인 실행과정을 계속해나간다. 그렇지 않으면 스핀잠그기가 정수값을 가진다고 판단할 때까지 표식 2에 있는(뒤붙이 b는 표식이 《앞》에 있음을 의미한다.) 순환을 실행한다. 그리고 나서 표식1부터 다시 시작하는데 그것은 다른 처리기가 잠그기를 획득했는가를 검사하지 않고 진행하는것은 위험하기때문이다. 펜티움4모형에서 등장한 pause 기호언어명령어는 스핀잠그기순환의 실행을 최적화하려고 만들었다. 약간 지연을 주어 잠그기이후에 나오는 코드의 실행을 빠르게 하고 전력소비를 줄인다.

pause명령어는 이전Intel처리기모형에서 아무일도 하지 않는 rep;nop명령어에 해당하기때문에 이전모형과 호환된다. spin_unlock마크로는 앞에서 획득한 스핀잠그기를 해제한다. 이것은 다음과 같은 코드를 만든다.

lock; movb $1,slp

다시 한번 lock앞붙이는 뒤에 나오는 값을 보관하는 명령어를 원자적으로 만든다.

5) 읽기/쓰기스핀잠그기

핵심부에서 동시에 할수 있는 작업량을 늘이기 위해 《읽기/쓰기스핀잠그기(read/write spin lock)》를 도입하였다. 이것은 자료구조체를 수정하는 핵심부조종경로가 없는 한 여러개의 핵심부조종경로에서 해당 자료구조체를 동시에 읽을수 있도록 허용한다.

핵심부조종경로에서 그 자료구조체에 쓰고싶을 때에는 읽기/쓰기잠그기중에서 쓰기용잠그기를 획득해야 하며 이것은 자원에 대한 배타적인 호출권한을 준다. 자료구조체를 동시에 읽을수 있게 하면 체계성능이 향상된다.

그림 8-2는 읽기/쓰기잠그기로 보호하는 두 제한령역(C1,C2)을 보여준다. 핵심부조종경로 R0과 R1이 동시에 C1에서 자료구조체를 읽고있으며 W0은 자료구조체의 쓰기용잠그기를 획득하려고 기다린다. 핵심부조종경로 W1은 C2에서 자료구조체를 쓰고있으며 R2와 W2는 매개의 자료구조체의 값을 읽거나 쓰기 위한 잠그기를 획득하려고 기

다린다.

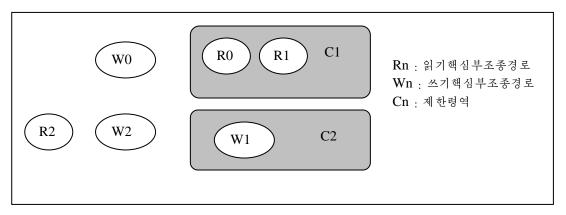



그림 8-2. 읽기/쓰기스핀잠그기

매 읽기/쓰기스핀잠그기는 rwlock_t구조체로 구성된다. 이 구조체의 lock마당은 다음과 같은 서로 다른 두가지 정보를 가지고있는 32bit값이다.(include\asm-(cpu종류)\spinlock.h)

・현재 보호중인 자료구조체를 읽는 핵심부조종경로의 수를 나타내는 24bit계수기. 이 마당의 0-23bit에 이 계수기의 2의 보수값이 들어간다.

・읽거나 쓰는 핵심부조종경로가 없을 때에는 1, 있을 때에는 0으로 되는 잠그기해제(unlock)기발. 이 기발은 이 마당의 비트24에 들어간다.

스핀잠그기를 리용하지 않을 때에는 lock마당의 값은 0x01000000이고(잠그기해제기발은 1이고 아무도 읽지 않는다.) 누군가 쓰기용으로 획득한 경우에는 0x00000000(잠그기해제기발은 0이고 읽지 않는다.)이다. 하나 또는 둘이상의 핵심부조종경로가 읽기용으로 획득한 경우에는0x00ffffff, 0x00fffffe 이런 식의 값을 가진다.(잠그기해제기발은 0이고 읽는 개수에 2의 보수를 취한값이 아래자리 24bit에 들어간다.)

rwlock_init마크로는 읽기/쓰기스핀잠그기의 lock마당을 0x01000000(열림)으로 초기화한다.(우와 같은 파일)

① 읽기용으로 잠그기를 획득하고 해제하기

읽기/쓰기스핀잠그기의 주소 rwlp에 대해 read_lock마크로를 실행하면 다음과 같은 코드로 바뀐다.

```
movl $rwlp,%eax
    lock; subl $1,(%eax)
    jns 1f
    call __read_lock_failed
1:
```

__read_lock_failed()는 다음기호언어함수이다.

```
__read_lock_failed:
     lock; incl(%eax)
1:   cmpl $1,(%eax)
     js 1b
     lock; decl(%eax)
     js __read_lock_failed
     ret
```

read_lock마크로는 앞서 설명한 spin_lock()마크로와 비슷한데 두번째 단계에서 다음의 함수를 실행한다.

```
int _raw_read_trylock(rwlock_t *lock)
{
      atomic_t *count = (atomic_t *)lock->lock;
      atomic_dec(count);
      if (atomic_read(count) >= 0)
          return 1;
      atomic_inc(count);
      return 0;
}
```

read_lock마크로는 원자적으로 스핀잠그기의 값을 1감소시켜 읽는 핵심부조종경로의 수를 증가시킨다. 감소시킨 연산의 결과가 부수가 아니라면 스핀잠그기를 획득한다. 부수라면 __read_lock_failed()함수를 호출한다. 이 함수는 원자적으로 lock마당의 값을 증가시켜 read_lock 마크로가 수행한 연산을 취소하고 그 마당이 정수(1보다 크거나 같은 값)가 될 때까지 순환한다. 다음으로 __read_lock_failed()는 스핀잠그기를 다시 획득하려고 시도한다.(다른 핵심부조종경로가 cmpl 명령어 직후에 스핀잠그기를 쓰기용으로 획득했을수도 있다.)

읽기용잠그기를 해제하는것은 아주 간단한다. read_unlock마크로는 단순히 lock마당의 계수기값을 증가시켜 읽는 경로의 수를 감소시킨다. 따라서 이 마크로는 다음기호언어명령어로 바뀐다.

```
lock; incl rwlp
```

② 쓰기용으로 잠그기를 획득하고 해제하기

write_lock마크로는 우에서 설명한 spin_lock(), read_lock()와 같은 방법으로

구현한다. 구체적으로 핵심부선취가 지원되면 그것을 불가능하게 하고 _raw_write_trylock()를 호출하여 잠그기를 시도한다. _raw_write_trylock()는 다음과 같다.

```
int _raw_write_trylock(rwlock_t *lock)
{
    atomic_t *count = (atomic_t *)lock->lock;
    if (atomic_sub_and_test(0x01000000, count))
        return 1;
    atomic_add(0x01000000, count);
    return 0;
}
```

읽기/쓰기스핀잠그기의 주소 rwlp에 대해 write_lock함수를 호출하면 다음과 같은 코드로 바뀐다.

```
movl $rwlp,%eax
    lock; subl $0x01000000,(%eax)
    jz 1f
    call __write_lock_failed
1:
```

__write_lock_failed()는 다음기호언어함수이다.

```
__write_lock_failed:
    lock; addl $0x01000000,(%eax)
 1: cmpl $0x01000000,(%eax)
    jne 1b
    lock; subl $0x01000000,(%eax)
    jnz __write_lock_failed
    ret
```

write_lock마크로는 원자적으로 스핀잠그기의 값에서 0x01000000을 덜어서 잠그기해제기발을 0으로 지운다. 이렇게 던 연산의 결과가 0이라면 (아무도 읽지 않음)스핀잠그기를 획득하고 그렇지 않으면 __write_lock_failed()를 호출한다.

이 함수는 원자적으로 lock마당에 0x01000000을 더해서 write_lock마크로가 수행한 연산을 취소하고 스핀잠그기가 여유가 있을 때까지(lock마당이 0x01000000) 순환한다. 다음으로 __write_lock_failed()는 스핀잠그기를 다시 획득하려고 시도한다.(다른 핵심부조종경로가 cmpl명령어직후에 스핀잠그기를 획득했을수도 있다.)

쓰기용잠그기를 해제하는것은 아주 간단하다. write_unlock마크로는 단순히 lock

마당의 잠그기해제기발을 설정한다. 따라서 이 마크로는 다음기호언어명령어로 바뀐다.

```
lock; addl $0x01000000,rwlp
```

6) **대형읽기용잠그기**

읽기/쓰기스핀잠그기는 읽는 대상이 많으면서 쓰는 대상이 적은 자료구조체에 유용하다.

이것은 보호하는 자료구조체를 여러 대상이 동시에 읽을수 있게 하기때문에 일반스핀잠그기보다 편리하다. 그렇지만 언제든지 CPU가 읽기/쓰기스핀잠그기를 획득할 때마다 rwlock_t에 있는 계수기를 갱신해야 한다.

이후에 다른 CPU에서 rwlock_t자료구조체를 호출하면 두 처리기의 하드웨어캐쉬를 동기화해야 하기때문에 상당한 성능손실이 발생한다. 또한 나중에 CPU도 rwlock_t 자료구조체를 바꾸기때문에 이전CPU에 있는 캐쉬가 무효화되고 이전CPU가 잠그기를 해제할 때도 또 다른 성능손실이 발생한다. 간단히 말하면 읽는 CPU는 읽기/쓰기스핀잠그기자료구조체를 담고있는 캐쉬행과 주고받으며 탁구를 치는것과 같다.

이런 문제를 해결하기 위하여 《대형읽기용읽기/쓰기스핀잠그기(big reader read/write spin lock)》이라는 특별한 종류의 읽기/쓰기스핀잠그기를 도입하였다.

그 기본사상은 잠그기의 《읽는》 부분을 모든 CPU로 분할하자는것이다. 따라서 매 CPU별로 자료구조체는 자기만의 하드웨어캐쉬행에 들어간다. 읽는 CPU는 다른 CPU에 있는 읽기용잠그기를 《상관하지》않으므로 서로 충돌할 필요가 없다. 반대로 한개의 CPU만이 쓰기용으로 잠그기를 획득할수 있기때문에 잠그기가 《쓰는》 부분은 모든 CPU에서 공유한다.

__brlock_array배렬은 대형읽기용읽기/쓰기스핀잠그기의 《읽는》 부분을 보관한다. 이 배렬은 이 종류의 모든 잠그기마다 그리고 체계에 있는 모든 CPU마다 하나씩 잠그기기발을 가진다. 해당 CPU가 읽기용도로 잠그기를 잠그면 이 기발의 값은 1이 된다. 반면에 __br_write_lock배렬은 대형읽기용스핀잠그기의 《쓰는》부분, 즉 대형읽기용스핀잠그기를 쓰기용으로 획득하면 설정하는 일반스핀잠그기를 보관한다.

br_read_lock()함수는 읽기용으로 스핀잠그기를 획득한다. 이 함수는 해당 CPU의 __brlock_array에 있는 잠그기기발과 대형읽기용스핀잠그기를 1로 설정한다. 그리고 __br_write_locks에 있는 스핀잠그기가 열릴 때까지 기다린다. 반대로 br_read_unlock()함수는 간단히 __brlock_array의 잠그기기발을 0으로 지운다.

br_write_lock()함수는 쓰기용으로 스핀잠그기를 획득한다. 이 함수는 spin_lock를 호출하여 대형읽기용스핀잠그기에 해당하는 __br_write_locks에 있는 스핀잠그기를 획득한 후 대형읽기용잠그기에 해당하는 __brlock_array에 있는 모든 잠그기기발이 0인가를 검사한다.

0이 아닌 잠그기기발이 하나라도 있으면 __br_write_locks에 있는 스핀잠그기를 해제하고 다시 시작한다.

br_write_unlock()함수는 간단히 spin_unlock를 호출하여 __br_write_locks에

있는 스핀잠그기를 해제한다. 열려있는 대형읽기용스핀잠그기를 읽기용으로 획득하는것은 매우 빠르다. 그러나 이것을 쓰기용으로 획득하려면 CPU는 스핀잠그기를 획득하고 여러개의 잠그기기발을 검사해야 하므로 매우 느리다. 따라서 대형읽기용스핀잠그기는 거의 사용하지 않는다. Intel기본방식에서는 망코드에서 대형읽기용스핀잠그기를 단 하나만 사용한다.

7) 신호기

앞에서 《동기화와 제한령역》에서 신호기(semaphore)를 설명하였다. 기본적으로 신호기는 자원을 기다리는측이 원하는 자원을 사용할수 있게 될 때까지 잠들게 하는 잠그기기법을 실현한다. 실제로 Linux는 두가지 종류의 신호기를 제공한다.

- 핵심부조종경로에서 사용하는 핵심부신호기
- 사용자방식프로세스가 사용하는 System V IPC신호기

IPC신호기는 다음절에서 설명하므로 이 절에서는 핵심부신호기에 대해서만 설명한다.

핵심부신호기(kernel semaphore)는 잠그기가 열리지 않는 한 핵심부조종경로가 더는 진행할수 없다는 점에서 스핀잠그기와 비슷하다. 그렇지만 핵심부조종경로가 핵심부신호기로 보호하는 이미 사용중인 자원을 획득하려고 하면 해당 프로세스를 보류한다. 해당 자원을 해제하면 그 프로세스는 다시 실행할수 있게 된다. 따라서 핵심부신호기는 프로세스가 잠들어도 되는 함수에서만 획득할수 있으며 새치기조종기나 지연함수에서는 사용할수 없다.

핵심부신호기는 다음과 같은 마당을 포함하는 struct semaphore형객체이다.

- count

atomic_t값을 보관한다. 이 값이 0보다 크면 자원은 자유롭다. 즉 자원은 현재 사용가능하다. 반대로 count가 0이면 신호기는 사용중이지만 보호하는 자원을 기다리는 다른 프로세스는 없다. 마지막으로 count가 부수이면 자원을 사용할수 없으며 적어도 하나이상의 프로세스가 그 자원을 기다리고있다.

- wait

현재자원을 기다리며 잠들어있는 모든 프로세스를 포함하는 대기렬목록의 주소를 보관한다. count가 0보다 크거나 같으면 대기렬은 비여있다.

- sleepers

프로세스가 신호기를 기다리며 잠들어있는가를 나타내는 기발을 보관한다. 배타적인 호출을 위한 신호기를 초기화할 때에는 init_MUTEX와 init_MUTEX_LOCKED마크로를 사용할수 있다. 이것들은 count마당을 각각 1(배타적인 호출을 하는 사용할수 있는 자원)과 0(현재신호기를 초기화하는 프로세스에 할당된 배타적인 호출을 하는 사용중인 자원)으로 설정한다. 참고로 신호기의 count값은 임의의 정수값 0으로 초기화할수 있다. 이 경우 최대 n개의 프로세스가 동시에 자원을 사용할수 있다.

① 신호기를 획득하고 해제하기

먼저 신호기를 해제하는 방법을 보자. 신호기를 획득하는것보다 해제하는것이 훨씬 간단하다. 프로쎄스는 up()기호언어함수를 호출하여 핵심부신호기잠그기를 해제한다.

이 함수는 다음과 같다.

```
movl $sem,%ecx
        lock; incl (%ecx)
        jg 1f
        pushl %eax
        pushl %edx
        pushl %ecx
        call __up_wakeup
        popl %ecx
        popl %edx
        popl %eax
1:
```

__up()은 다음과 같은 C함수이다.

```
__attribute__((regparm(3))) void _ _up(struct semaphore *sem)
{
wake_up(&sem->wait);
}
```

up()함수는 신호기 *sem의 count마당(semaphore구조체의 편위0에 있다.)을 증가시킨 후 그 값이 0보다 큰가를 검사한다. count값을 증가시키고 다음행에 나오는 이행(jump)명령어에서 검사할수 있도록 기발을 설정하는 작업은 반드시 원자적으로 실행해야 한다. 그렇지 않으면 다른 핵심부조종경로가 동시에 그 마당의 값에 호출하여 뜻밖의 결과를 낳을수 있다. count가 0보다 크다면 대기렬에서 기다리는 프로쎄스가 없다는것이므로 더는 할 일이 없다. 0보다 같거나 작으면 __up()함수를 호출하여 잠든 한개의 프로쎄스를 깨운다.

프로쎄스는 핵심부신호기잠그기를 획득할 때 down()함수를 호출한다. 이 함수는 복잡하지만 기본적으로 다음의 코드와 동등하다.

```
down:
        movl $sem, %ecx
        lock; decl (%ecx);
        jns 1f
```

```
        pushl %eax
        pushl %edx
        pushl %ecx
        call __down
        popl %ecx
        popl %edx
        popl %eax
1:
```

__down()은 다음과 같은 C함수이다.

```
    __attribute__((regparm(3))) void _ _down(struct semaphore * sem)
    {
        DECLARE_WAITQUEUE(wait, current);
        unsigned long flags;
        current->state = TASK_UNINTERRUPTIBLE;
        spin_lock_irqsave(&sem->wait.lock, flags);
        add_wait_queue_exclusive_locked(&sem->wait, &wait);
        sem->sleepers++;
        for (;;) {
            if (!atomic_add_negative(sem->sleepers-1, &sem->count)) {
                sem->sleepers = 0;
                break;
            }
            sem->sleepers = 1;
            spin_unlock_irqrestore(&sem->wait.lock, flags);
            schedule( );
            spin_lock_irqsave(&sem->wait.lock, flags);
            current->state = TASK_UNINTERRUPTIBLE;
        }
        remove_wait_queue_locked(&sem->wait, &wait);
        wake_up_locked(&sem->wait);
        spin_unlock_irqrestore(&sem->wait.lock, flags);
        current->state = TASK_RUNNING;
    }
```


교육성 교육정보쎈터

down()함수는 먼저 신호기 *sem의 count마당(semaphore구조체의 편위0에 있다.)을 감소시킨 후 그 결과가 부수인가를 검사한다. 마찬가지로 값을 감소시키고 검사하는 작업은 원자적으로 실행해야 한다. count가 0보다 크거나 같으면 현재프로쎄스는 자원을 획득하여 정상적으로 실행을 계속한다. count가 부수이면 현재프로쎄스의 실행을 보류해야 한다.

이때에는 일부등록기의 내용을 탄창에 보관한 후 __down()을 호출한다.

기본적으로 __down()함수는 현재프로쎄스의 상태를 TASK_RUNNING에서 TASK_UNINTERRUPTTBLE로 바꾸고 프로쎄스를 신호기의 대기렬에 넣는다. semaphore구조체의 다른 마당에 호출하기 전에 semaphore_lock스핀잠그기와 국부새치기를 금지한다. 이것은 현재프로쎄스가 신호기에 있는 마당을 갱신하는 동안에 다른 CPU에서 실행중인 프로쎄스가 읽거나 수정할수 없게 한다. __down()함수의 기본역할은 신호기를 해제할 때까지 현재프로쎄스를 보류하는것이다. 그렇지만 이 일을 처리하는 방법은 복잡하다. 코드를 쉽게 리해하기 위해 신호기의 sleepers마당은 보통신호기의 대기렬에서 잠든 상태에 있는 프로쎄스가 없으면 0, 있으면 1이라고 생각하자. 몇가지 전형적인 실례를 보면서 이 코드를 설명해보자.

○ 뮤텍스(Mutex)열린신호기(count는 1이고 sleepers는 0이다.)

down마크로는 단순히 count마당을 0으로 설정하고 프로그람의 다음명령어로 이행한다. 따라서 __down()함수를 전혀 실행하지 않는다.

○뮤텍스닫긴신호기, 기다리는 프로쎄스 없음(count는 0이고 sleepers는 0이다.)

down마크로는 count를 감소시키고 count마당이 -1이고 sleepers마당이 0인 상태에서 __down()함수를 호출한다.

이 함수는 순환을 반복할 때마다 count마당이 부수인가를 검사한다.(이 함수를 호출할 때 sleepers는 0이므로 atomic_add_negative()에 의해 마당이 변경되지 않는다.)

・count마당이 부수이면 schedule()함수를 호출하여 현재프로쎄스를 보류한다. count마당은 여전히 -1이고 sleepers마당은 1이 된다.

프로쎄스는 련이어 순환안에서 실행을 시작하고 검사를 다시 한다.

・count마당이 부수가 아니면 sleepers를 0으로 설정하고 순환에서 빠져나간다.

신호기대기렬에 있는 다른 프로쎄스를 깨우고 (그러나 현재 대기렬은 비여있다.) 신호기를 가진채로 완료한다. 완료할 때 신호기가 닫겨있고 기다리는 프로쎄스가 없다는것을 나타내도록 count마당과 sleepers마당을 모두 0으로 설정한다.

○ 뮤텍스닫긴신호기, 기다리는 프로쎄스 있음(count는 -1이고 sleepers는 1이다.)

down마크로는 count를 감소시키고 count마당이 -2이고 sleepers 마당이 1인 상태에서 __down()함수를 호출한다. 이 함수는 림시적으로 sleepers를 2로 설정하고 나

서 count에 sleepers-1을 더해서 down 마크로가 수행한 감소명령을 취소한다. 이와 동시에 count가 여전히 부수인가를 검사한다.(__down()이 제한령역에 들어가기 직전에 신호기를 가지고있던 프로쎄스가 이것을 해제했을수 있다.)

·count마당이 부수이면 sleepers를 1로 설정하고 schedule()를 호출하여 현재프로쎄스를 보류한다. count마당은 여전히 -1이고 sleepers마당은 1이다.

·count마당이 부수가 아니면 sleepers를 0으로 설정하고 신호기대기렬에 있는 다른 프로쎄스를 깨운 후 신호기를 가진채 완료한다. 완료할 때 count마당과 sleepers 마당은 모두 0이다.

다른 잠든 상태에 있는 프로쎄스가 《있기》때문에 이 값을 둘다 틀렸다고 생각할수도 있다.

그렇지만 대기렬에 있는 다른 프로쎄스가 이미 깨여났다는것을 기억하자. 이 프로쎄스는 순환을 다시 반복한다. atomic_add_negative()함수는 count에서 1을 덜고 이것을 다시 -1로 만든다. 나가서 잠든 상태로 돌아가기 전에 이 깨여난 프로쎄스는 sleepers를 1로 재설정한다.

우에서 본것처럼 이 코드는 모든 경우에 정확히 동작한다. 대기렬에 있는 잠든 프로쎄스는 배타적이기때문에(《 프로쎄스조직화하기 》참고) __down()함수에서 호출하는 wake_up()함수는 한개의 프로쎄스만 깨운다. 례외조종기만이 체계호출봉사루틴에서 down()함수를 사용할수 있다.

down()함수는 신호기가 이미 사용중이면 프로쎄스를 보류할수 있으므로 새치기조종기나 지연함수는 이것을 호출해서는 안된다. 이런 리유로 Linux는 앞에서 설명한것처럼 비동기적인 함수에서 안전하게 사용할수 있는 down_trylock()이라는 함수를 제공한다.

이 함수는 자원이 사용중일 때 프로쎄스를 재우지 않고 곧바로 완료한다는 점을 제외하면 down()함수와 똑같다.

이와 조금 다른 함수로 down_interruptible()이 있다. 이 함수는 신호기에 의해 차단상태가 되더라도 프로쎄스가 신호를 받아 《down》동작을 멈출수 있게 하므로 장치구동프로그람에서 많이 사용한다. 잠든 상태에 있는 프로쎄스가 필요한 자원을 획득하기 전에 신호를 받아서 깨여나면 이 함수는 신호기의 count마당값을 증가시킨 후 -EINTR값을 되돌린다.

반대로 down_interruptible()함수가 정상적으로 실행을 완료하여 자원을 획득하면 0을 되돌린다. 따라서 장치구동프로그람은 결과값이 -EINTR인 경우 입출력동작을 멈춘다.

마지막으로 프로쎄스가 신호기를 획득하려 할 때 대체로 신호기는 열린 상태에 있기때문에 신호기함수는 이 경우에 맞추어 최적화되여있다. 특히 up()함수는 신호기대기렬이 비여있다면 제한령역에 들어가지 않는다. 비슷하게 down()함수는 신호기가 열린 상

태이면 제한령역에 들어가지 않는다. 신호기을 실현하는 복잡한 부분은 실행흐름의 핵심 부분에서 시간이 소비되는 명령어를 피하도록 노력한 결과이다.

8) 읽기/쓰기신호기

읽기/쓰기신호기는 Linux의 새로운 기능이다. 이것은 신호기를 다시 사용할수 있게 될 때까지 기다리는 프로쎄스를 보류한다는 점을 제외하면 앞서 《읽기/쓰기스핀잠그기》에서 설명한 읽기/쓰기스핀잠그기와 비슷하다.

많은 핵심부조종경로는 동시에 읽기/쓰기잠그기를 읽기용으로 획득할수 있지만 쓰려는 핵심부조종경로는 보호하는 자원에 대한 배타적인 호출을 해야 한다. 따라서 읽기나 쓰기호출을 위해 신호기를 가지는 다른 핵심부조종경로가 없을 때에만 쓰기용으로 신호기를 획득할수 있다.

읽기/쓰기신호기는 핵심부내부에서 동기정도를 향상하고 전체적인 체계성능도 향상한다. 핵심부는 읽기/쓰기신호기를 기다리는 모든 프로쎄스를 엄격하게 FIFO순서에 따라 처리한다.

신호기가 잠겨있어 기다려야 하는 읽기 또는 쓰기하려는 프로쎄스를 신호기의 대기렬목록의 맨끝에 추가한다. 신호기를 해제하면 대기렬의 맨 처음위치에 있는 프로쎄스를 검사하여 항상 맨 처음에 있는 프로쎄스를 깨운다. 그것이 쓰려는 프로쎄스이면 대기렬에 있는 다른 프로쎄스는 계속 잠들어 있어야 한다. 읽으려는 프로쎄스이면 처음 프로쎄스뒤에 나오는 다른 읽으려는 프로쎄스도 깨여나서 잠그기를 획득한다. 그렇지만 대기렬에서 쓰려는 프로쎄스뒤에 있는 읽으려는 프로쎄스는 계속 잠든 상태로 있는다.

매 읽기/쓰기신호기는 아래와 같은 마당을 포함하는 rw_semaphore구조체로 나타낸다.

• count

두개의 16bit계수기를 보관한다. 웃자리 16bit계수기에는 대기중이 아닌 쓰려는 프로쎄스의 개수와(0이나 1) 대기중인 핵심부조종경로개수의 합에 2의 보수를 취한 값이 들어간다. 아래자리 16bit계수기에는 대기중이 아닌 읽거나 쓰려는 프로쎄스의 전체 개수가 들어간다.

• wait_list

기다리는 프로쎄스목록에 대한 지적자이다. 이 목록에 있는 매 요소는 rwsem_waiter구조체로서 잠들어있는 프로쎄스의 서술자를 가리키는 지적자와 프로쎄스가 읽기용신호기를 원하는가 쓰기용을 원하는가를 나타내는 기발을 포함한다.

• wait_lock

대기렬목록과 rw_semaphore구조체자신을 보호하기 위한 스핀잠그기이다. init_rwsem()함수는 rw_semaphore구조체의 count마당을 0으로 설정하고 wait_lock 스핀잠그기를 열린 상태로 wait_list를 빈 목록으로 초기화한다. down_read()와 down_write()함수는 각각 읽기용과 쓰기용으로 읽기/쓰기신호기를 획득한다.

up_read()(실지는 __up_read()함수를 호출)와 up_write()(실지는

__up_write()함수를 호출)함수도 이전에 읽기와 쓰기용으로 획득한 신호기를 해제한다.

9) 완료

Linux는 신호기와 비슷한 《완료(completion)》라는 동기화기법도 사용하는데 다중처리기체계에서 다음과 같은 경우에 발생하는 미묘한 경쟁상태를 해결할 목적으로 도입하였다.

프로쎄스 A는 림시신호기변수를 할당해서 닫긴 뮤텍스(MUTEX)로 초기화하고 이것의 주소를 프로쎄스 B에 전달한 후 신호기에 down()함수를 호출한다. 나중에 다른 CPU에서 실행하는 프로쎄스 B는 같은 신호기에 대해 up()를 실행한다. 그런데 현재 실현된 up()와 down()함수를 같은 신호기에 대해서 동시에 실행하게 된다.

따라서 프로쎄스 A가 깨여나서 림시신호기를 제거할 때 프로쎄스 B는 아직 up()함수를 실행하고있을수도 있다. 그 결과 up()는 존재하지 않는 자료구조체를 호출할수도 있다. 물론 down()과 up()함수의 실현을 바꿔서 같은 신호기에 대해서 두 함수를 동시에 실행하지 못하게 만들수도 있다. 그렇지만 이렇게 변경하려면 명령어를 추가해야하고 이 함수를 많이 사용하기때문에 체계성능이 떨어질수 있다.

완료는 이 문제를 해결하기 위하여 특별히 도입한 동기화기법이다. completion은 대기렬머리부와 기발 하나를 포함하는 자료구조체이다.(completion.h 참고)

```
struct completion {
    unsigned int done;
    wait_queue_head_t wait;
};
```

up()에 대응되는 함수는 complete()이다. 이 함수는 completion자료구조체에 대한 주소를 파라메터로 받아서 이것의 done마당을 1 증가하고 __wake_up_common()을 호출하여 대기렬 wait에서 잠들어 있는 배타적인 프로쎄스를 깨운다.

down()에 대응되는 함수는 wait_for_completion()이다. 이 함수는 completion자료구조체에 대한 주소를 파라메터로 받아서 done기발의 값을 검사한다. 이것이 1이상이라면 다른 CPU에서 complete()함수를 이미 실행했으므로 wait_for_completion()함수는 완료한다. 0이라면 current를 대기렬의 끝에 배타적인 프로쎄스로 추가하고 current를 TASK_UNINTERRUPTTBLE상태로 만들어 잠들게 한다. complete()함수에 의해 깨여나면 current를 대기렬에서 제거하고 done마당을 0으로 설정한 후 완료한다.

완료와 신호기의 실질적인 차이는 대기렬에 들어있는 스핀잠그기를 사용하는 방법이다.

complete()와 wait_for_completion()함수는 이것들을 동시에 실행하지 않는다는 것을 보장하기 위해 스핀잠그기를 사용하지만 up()와 down()는 대기렬목록에 직렬로 호출하기 위하여 스핀잠그기를 사용한다.

10) 국부새치기금지

새치기금지는 일련의 핵심부코드를 제한령역으로 다루게 하는 핵심기구중의 하나이다. 이것은 하드웨어장치가 IRQ신호를 발생시키더라도 핵심부조종경로는 작업을 계속 진행할수 있게 한다. 따라서 새치기조종기에서도 호출하는 자료구조체를 효과적으로 보호하는 방법을 제공한다. 그러나 국부새치기금지(local interrupt disabling)만으로 다른 CPU에서 실행하는 새치기조종기에서 동시에 자료구조체를 호출하는것까지는 막을수 없다. 따라서 다중처리기체계에서는 일반적으로 국부새치기금지를 스핀잠그기와 함께 사용한다.(《핵심부자료구조체로의 호출동기화》참고)

cli기호언어명령어로 한 CPU에서 새치기를 금지할수 있다. __cli()마크로는 이 명령어를 만든다. sti기호언어명령어로 한 CPU에서 새치기를 허용할수 있다. __sti()마크로는 이 명령어를 만든다.

최근핵심부에서는 local_irq_disable()과 local_irq_enable()이라는 마크로도 정의한다. 이것들은 각각 __cli()와 __sti()와 같지만 이름이 기본방식에 의존적이지 않고 훨씬 리해하기 쉽다.

핵심부가 제한령역에 들어갈 때 eflags 등록기의 IF기발을 0으로 지워서 새치기를 금지한다.

그러나 제한령역에서 나갈 때 단순히 기발을 1로 다시 설정해서는 안된다. 새치기는 동시에 실행될수도 있기때문에 핵심부는 현재핵심부조종경로를 실행하기 전에 IF기발의 값이 무엇이였는가를 알수 없다. 이런 경우에 핵심부조종경로는 이전기발설정을 보관하였다가 끝날 때 이 기발값을 회복해야 한다. __save_flags와 __restore_flags마크로를 사용하여 eflags등록기의 내용을 보관하고 회복할수 있다.

다음은 이 마크로를 사용하는 전형적인 방법이다.

```
__save_flags(old);
__cli();
[…]
__restore_flags(old);
```

__save_flags 마크로는 eflags등록기내용을 국부변수old로 복사하고 __cli()는 IF기발을 0으로 지운다. 제한령역끝에서는 __restore_flags 마크로가 eflags등록기를 이전 값으로 회복한다. 따라서 이 조종경로가 __cli()마크로를 호출하기 전에 새치기를 허용한 상태였을 때에만 새치기를 허용한다. 최근 핵심부에서는 local_irq_save()와 local_irq_restore()라는 마크로도 정의한다. 이것들은 각각 __save_flags()와 __restore_flags()와 같지만 이름이 훨씬 리해하기 쉽다.

11) 대역새치기금지

몇가지 중요한 핵심부함수는 다른 CPU에서 새치기조종기나 지연함수를 실행하지 않을 때에만 실행할수 있다. 《대역새치기금지(global interrupt disabling)》는 이런 동기화요구를 해결한다. 이런 전형적인 경우로 하드웨어장치를 재설정(reset)해야 하는

구동프로그람이 있다.

구동프로그람은 입출력포구에 어떤 일을 하기 전에 대역새치기를 금지해서 다른 구동프로그람이 같은 포구를 호출하지 못하게 해야 한다.

이 절에서 보지만 대역새치기금지는 체계의 동시성수준을 크게 낮춘다. 이것을 다른 효률적인 동기화기법으로 교체할수 있으므로 이것을 사용하는것은 불합리하다.

0	__global_cli()
1	__global_sti()
2	__cli()
3	__sti()

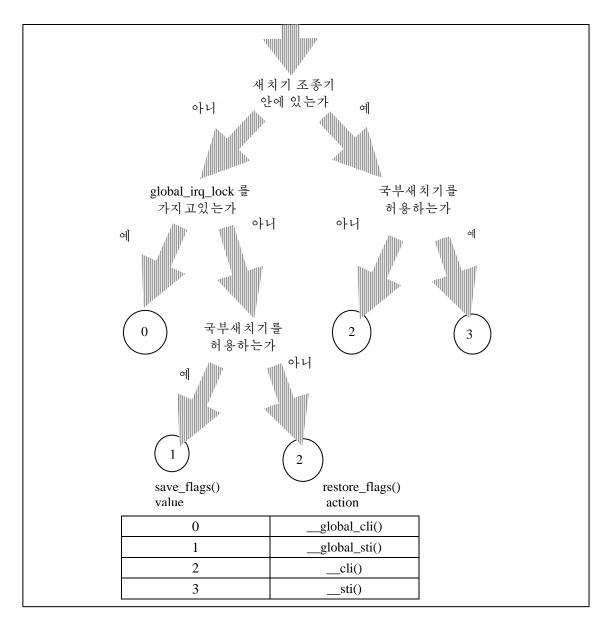



그림 8-3. save_flags()와 restore_flags 가 하는 일

12) 지연함수금지

《프로그람적인 새치기》에서 지연함수를 실행할 시점은 예측할수 없다고 설명하였다.(기본적으로 하드웨어새치기조종기를 완료할 때이다.) 따라서 지연함수에서 호출하는 자료구조체를 경쟁상태로부터 보호해야 한다.

어떤 CPU에서 지연함수의 실행을 금지하는 가장 쉬운 방법은 해당 CPU의 새치기를 금지하는것이다.

새치기조종기를 실행하지 않으면 프로그람적인 새치기(softirq)작업 역시 비동기적으로 시작하지 않는다. 모든 CPU에서 대역새치기를 금지한다면 모든 CPU에서 지연함수의 실행을 금지할수 있다.

cli()마크로를 실행하면 이것을 호출한 핵심부조종경로는 모든 CPU가 지연함수를 실행하지 않고있으며 대역새치기를 허용하기 전까지 새로 시작하지 않는다고 생각할수 있다.

그러나 후에 보지만 핵심부는 어떤 경우에 새치기를 금지하지 않은채로 지연함수를 금지할 필요가 있다.

매 CPU마다 지연함수를 금지하려면 CPU에 관련된 irq_cpustat_t구조체의 __local_bh_count마당을 0이 아닌 값으로 설정하면 된다. do_softirq()함수는 이 값이 0이 아닌 경우 프로그람적인 새치기를 절대로 실행하지 않는다.

local_bh_disable마크로는 __local_bh_count값을 1 증가시키고 local_bh_enable마크로는 이 값을 1감소시킨다. 따라서 핵심부는 local_bh_disable를 여러번 호출할수 있다. 처음 호출한 local_bh_disable과 대응되는 local_bh_enable마크로를 호출한 경우에만 지연함수를 다시 허용한다.

3. 핵심부자료구조체로의 호출동기화

앞에서 설명한 몇가지 동기화기법을 리용하여 공유하는 자료구조체를 경쟁상태로부터 보호할수 있다.

어떤 종류의 동기화기법을 사용하는가에 따라 체계성능이 현저하게 달라질수 있다. 일반적으로 핵심부개발자들은 항상 동시성수준을 체계에서 가능한 가장 높은 수준으로 유지하는것을 제일가는 규칙으로 삼는다.

체계의 동시성수준(concurrency level)은 다음 두가지 요인에 의하여 좌우된다.

- 동시에 동작하는 입출력장치의 개수
- 과제를 처리하는 CPU의 개수

입출력처리량을 늘이려면 새치기를 금지하는 시간이 매우 짧아야 한다. 《IRQ와 새치기》에서 설명한것처럼 새치기를 금지하고있을 때에는 입출력장치가 IRQ를 발생시켜도 PIC는 이것을 일시적으로 무시하여 이 장치에 대해서 새로운 작업을 시작할수 없다.

CPU를 효률적으로 사용하려면 스핀잠그기를 리용한 동기화기법은 가능한 피해야 한다.

CPU가 스핀잠그기가 열리기를 기다리며 명령어순환을 실행하는것은 해당 장치의 귀중한 시간을 랑비하는것이다.

동시성수준을 높게 유지하면서 동기화를 할수 있는 몇가지 경우를 보면 다음과 같다.

·정수값 하나로 구성된 공유자료구조체를 갱신할 때에는 이 자료구조체를 atomic_t형으로 선언하여 원자적인 연산을 사용할수 있다. 원자적인 연산은 스핀잠그기와 새치기금지보다 빠르며 동시에 해당 자료구조체를 호출하는 핵심부조종경로만 느리게 만들뿐이다.

· 공유하는 련결목록에 항목을 추가하는것을 지적자를 적어도 두개 설정해야 하기때문에 절대로 원자적이지 않다. 그럼에도 불구하고 핵심부는 때때로 잠그기를 사용하거나 새치기를 금지하지 않고도 이 삽입연산을 수행할수 있다. 이렇게 동작하는 실례로 체계호출봉사루틴이(《체계호출조종기와 봉사루틴》참고) 단순련결목록에 새 항목을 삽입할 때와 새치기조종기나 지연함수에서 이 목록을 비동기적으로 읽기만 하는 경우이다.

C언어에서는 다음지적자를 할당하는 행을 통해 삽입연산을 진행한다.

```
new->next = list_element->next;
list_element->next = new;
```

이 삽입연산은 기호언어에서 련속된 원자적인 명령어 두개로 바뀐다.

첫번째 명령어는 new항목의 next지적자를 설정하지만 목록을 변경하지는 않는다. 따라서 첫번째 명령어와 두번째 명령어사이에서 새치기조종기를 실행하여 목록을 참조한다면 새 항목이 들어가지 않은 목록을 보게 된다.

두번째 명령어가 실행한 후에 새치기조종기를 실행하면 새 항목이 들어간 목록을 보게 된다. 중요한 점은 두 경우 모두 목록는 정상이며 손상되지 않은 상태라는것이다. 그러나 이것은 새치기조종기가 목록을 고치지 않는 경우에만 해당한다.

목록을 수정하면 new항목의 new지적자에 설정한 값이 잘못될수도 있다. 개발자는 콤파일러나 CPU의 조종장치가 두개의 대입연산의 순서를 바꾸어 놓지 않도록 해야 한다. 그렇지 않으면 체계호출루틴의 두개의 대입연산사이에 새치기조종기를 실행하는 경우에 조종기는 손상된 목록을 참조하게 된다.

따라서 쓰기용기억기장벽이 필요하다.

```
new->next = list_element->next;
wmb();
list_element->next = new;
```

1) 스핀잠그기와 신호기, 새치기금지가운데서 선택하기

대부분의 핵심부조종구조체를 호출하는 형태는 우에서 본 간단한 실례보다도 훨씬 더 복잡하여 핵심부개발자들은 신호기와 스핀잠그기, 새치기금지,프로그람적새치기금지를 사용해야 한다.

일반적으로 어떤 동기화기법을 사용하겠는가는 표 8-7에 나타낸것처럼 어떤 종류의 핵심부조종경로가 그 자료구조체를 호출하는가에 달려있다.

표 8-7. 핵심부조종경로에서 자료구조를 호출할 때 필요한 보호

자료구조를체를 호출하는 핵심부조종경로	UP에서 필요한 보호	MP에서 추가로 필요한 보호
례외	신호기	없음
새치기	국부새치기금지	스핀잠그기
지연함수	없음	없거나 스핀잠그기 (표 8-8참고)
례외+새치기	국부새치기금지	스핀잠그기
례외+지연함수	국부프로그람적인 새치기금지	스핀잠그기
새치기+지연함수	국부새치기금지	스핀잠그기
례외+새치기+지연함수	국부새치기금지	스핀잠그기

이 표에 대역새치기금지는 없다. 모든 CPU에서 새치기를 지연시키는것은 체계의 동시성수준을 현저히 낮추기때문에 일반적으로 대역새치기금지를 피하고 그대신 다른 동기화기법을 사용해야 한다. 사실 대역새치기금지는 Linux 2.4에서 이전장치구동프로그람을 지원하기 위하여 남아 있지만 현재개발판본인 Linux 2.6에서는 이것을 삭제하였다.

○ **례외에서 호출하는 자료구조체보호**

자료구조체를 례외조종기에서만 호출한다면 경쟁상태가 일어나는 경우를 쉽게 리해하고 방지할수 있다.

동기화문제를 일으키는 가장 일반적인 례외는 사용자방식프로그람에 봉사를 제공하려고 CPU가 핵심부방식으로 바뀌여 실행하는 체계호출봉사루틴이다.(《체계호출조종기와 봉사루틴》참고) 따라서 례외에서만 호출하는 자료구조체는 대체로 하나이상의 프로쎄스에 할당할수 있는 자원을 나타낸다.

신호기는 자원을 사용할수 있을 때까지 프로쎄스를 잠재울수 있으므로 이 기법을 리용하여 경쟁상태를 막는다. 신호기는 단일처리기체계과 다중처리기체계에서 모두 똑같이 동작한다.

○ **새치기에서 호출하는 자료구조체보호**

어떤 자료구조체를 한 새치기조종기에서만 호출한다고 하자. 《새치기처리》에서 매 새치기조종기를 자기 자신에 대해 직렬화한다고 설명하였다. 즉 같은 새치기조종기를 둘이상 동시에 실행할수 없다. 따라서 자료구조체를 호출할 때 어떤 동기화기법도 필요없다.

그렇지만 여러개의 새치기조종기에서 호출하는 자료구조체라면 사정이 달라진다.

새치기조종기를 다른 조종기가 가로챌수도 있고 다중처리기체계에서는 서로 다른 새치기조종기를 동시에 실행할수도 있다. 동기화를 하지 않으면 공유하는 자료구조체가 쉽게 파괴될수 있다. 단일처리기체계에서는 새치기조종기의 모든 제한구역에서 새치기를 금지하여 경쟁상태를 막을수 있다.

다른 동기화기법으로는 이 작업을 완수할수 없으므로 다른 방법이 없다. 신호기는 프로쎄스를 차단할수 있기때문에 새치기조종기에서 사용할수 없다.

반면에 스핀잠그기를 사용하면 자료구조체를 호출하는 새치기조종기를 가로챈 새로운 새치기조종기는 스핀잠그기가 열리기를 기다리며 순환을 반복하여 이전 조종기가 잠그기를 해제하지 못해 체계가 정지되게 된다. 일반적으로 다중처리기체계에서는 더 많은 요구이 존재한다.

국부새치기를 금지하는것만으로 경쟁상태를 막을수 없다. 사실 어느 CPU에서 새치기를 금지하더라도 다른 CPU에서 새치기를 계속 실행할수 있다.

여기서 경쟁상태을 막는 가장 편리한 방법은 국부새치기를 금지하고 (그래서 이것을 실행한 같은 CPU에서 다른 새치기조종기가 현재조종기를 가로채지 못하게 하고) 그 자료구조체를 보호하는 스핀잠그기이나 읽기/쓰기스핀잠그기를 획득하는것이다.

이렇게 추가로 스핀잠그기를 사용하는 경우 한 새치기조종기가 사용하려는 스핀잠그기가 잠긴상태라고 해도 잠그기를 소유하는 다른 CPU에서 실행중인 새치기조종기가 잠그기를 해제할수 있으므로 체계가 정지되는 일은 없다.

Linux핵심부는 새치기를 허용/금지하는 일과 스핀잠그기를 다루는 일을 같이 하는 여러가지 마크로를 사용한다.

표 8-8은 이런 일을 수행하는 모든 마크로를 보여준다. 단일처리기체계에서는 스핀잠그기를 다루는 마크로가 아무 일도 하지 않으므로 이 마크로들은 국부새치기만을 허용하거나 금지한다.

표 8-8. 새치기를 고려하는 스핀잠그기마크로

함 수	설 명
spin_lock_irq(l)	local_irq_disable();preempt_disable();_raw_spin_lock(l)
spin_unlock_irq(l)	_raw_spin_unlock(l);local_irq_enable();preempt_enable()
spin_lock_bh(l)	local_bh_disable( ); spin_lock(l)
spin_unlock_bh(l)	spin_unlock(l); local_bh_enable()

spin_lock_irqsave(I,f)	local_irq_save(f);preempt_disable();_raw_spin_lock(l)
spin_unlock_irqrestore(i,f)	_raw_spin_unlock(l);local_irq_restore(f);preempt_enable();
read_lock_irq(l)	local_irq_disable();preempt_disable();_raw_read_lock(l)
read_unlock_irq(l)	_raw_read_unlock(l);local_irq_enable();preempt_enable();
read_lock_bh(l)	local_bh_disable( ); read_lock(l)
read_unlock_bh(l)	read_unlock(l); local_bh_enable( )
write_lock_irq(l)	local_irq_disable();preempt_disable();_raw_write_lock(l)
write_unlock_irq(l)	_raw_write_unlock(l);local_irq_enable();preempt_enable()
write_lock_bh(l)	local_bh_disable( ); write_lock(l)
write_unlock_bh(l)	write_unlock(l); local_bh_enable( )
read_lock_irqsave(i,f)	local_irq_save(f); preempt_disable();_raw_read_lock(l)
read_unlock_irqrestore(i,f)	_raw_read_unlock(l);local_irq_restore(f);preempt_enable()
write_lock_irqsave(i,f)	local_irq_save(f); preempt_disable();_raw_write_lock(l)
write_unlock_irqrestore(i,f)	_raw_write_unlock(l);local_irq_restore(f);preempt_enable();
read_seqbegin_irqsave(l,f)	local_irq_save(f); read_seqbegin(l)
read_seqretry_irqrestore(l,v,f)	read_seqretry(l,v); local_irq_restore(f)
write_seqlock_irqsave(l,f)	local_irq_save(f); write_seqlock(l)
write_sequnlock_irqrestore(l,f)	write_sequnlock(l); local_irq_restore(f)
write_seqlock_irq(l)	local_irq_disable( ); write_seqlock(l)
write_sequnlock_irq(l)	write_sequnlock(l); local_irq_enable( )
write_seqlock_bh(l)	local_bh_disable( ); write_seqlock(l);
write_sequnlock_bh(l)	write_sequnlock(l); local_bh_enable( )

○ **지연함수에서 호출하는 자료구조체보호**

지연함수에서만 호출하는 자료구조체에는 어떤 보호가 필요하는가하는것은 대체로 지연함수의 류형에 따라 좌우된다. 《프로그람적 새치기와 소작업, 하반부》에서 프로그람적 새치기와 소작업, 하반부는 기본적으로 동시성의 정도가 다르다고 설명하였다.

무엇보다도 단일처리기체계에서는 경쟁상태가 있을수 없다. 한 CPU에서 지연함수를 항상 직렬로 실행하기때문이다. 즉 지연함수를 다른 지연함수가 가로챌수 없다. 따라서 동기화기법이 필요없다. 반대로 다중처리기체계에서는 여러 지연함수를 동시에 실행할수 있기때문에 경쟁상태가 실제로 존재한다. 표 8-9에서 모든 가능한 경우를 표시하였다.

표 8-9. SMP에서 지연함수가 자료구조체를 호출할 때 필요한 보호

자료구조체를 호출하는 지연함수	보호
프로그람적인 새치기	스핀잠그기
소작업 한개	없음
소작업 여러개	스핀잠그기
하반부	없음

같은 프로그람적인 새치기를 두개이상의 CPU에서 동시에 실행할수 있으므로 프로그람적인 새치기에서 호출하는 자료구조체를 스핀잠그기와 같은것을 리용하여 항상 보호해야 한다.

같은 종류의 소작업은 동시에 실행할수 없기때문에 한 종류의 소작업에서 호출하는 자료구조체는 보호할 필요가 없다. 그렇지만 여러개의 소작업이 호출하는 자료구조체인 경우는 보호해야 한다. 마지막으로 하반부는 절대로 동시에 실행하지 않으므로 하반부에서만 호출하는 자료구조체를 보호할 필요는 없다.

○ **례외와 새치기에서 호출하는 자료구조체보호**

례외(례를 들면 체계호출봉사루틴)와 새치기조종기에서 모두 호출하는 자료구조체를 보자.

새치기조종기는 재진입할수 없고 례외가 새치기를 가로챌수 없으므로 단일처리기체계에서는 아주 간단하게 경쟁상태을 막을수 있다.

핵심부가 국부새치기를 금지한 상태에서 자료구조체를 호출하는 동안에는 아무도 자료구조체를 호출하는 핵심부를 가로챌수 없다. 그렇지만 어떤 자료구조체를 한 종류의 새치기조종기에서만 호출한다면 새치기조종기는 국부새치기를 금지하지 않고도 자료구조체를 자유롭게 호출할수 있다.

다중처리기체계에서는 다른 CPU에서 동시에 실행하는 새치기와 례외를 주의해야 한다.

국부새치기금지를 스핀잠그기와 같이 사용하여 조종기가 자료구조체를 호출하는 일

을 끝마칠 때까지 동시에 실행하는 핵심부조종경로를 기다리게 만든다. 때때로 스핀잠그기를 신호기로 바꾸는것이 좋은 경우가 있다. 새치기조종기를 보류할수 없으므로 새치기조종기에서는 down_trylock()함수를 호출하는 순환을 반복하여 신호기를 획득해야 한다. 이렇게 하면 신호기는 본질적으로 스핀잠그기처럼 동작한다. 반면에 체계호출봉사루틴은 신호기가 사용중인 경우 자기를 호출한 프로쎄스를 보류할수 있다.

대부분의 체계호출에서는 이런 식으로 동작한다. 이것은 체계의 동시성수준을 높이기때문에 합리적이다.

○ 례외와 지연함수에서 호출하는 자료구조체보호

례외조종기와 지연함수에서 모두 호출하는 자료구조체는 례외와 새치기조종기에서 호출하는 자료구조체처럼 다룰수 있다. 사실 지연함수는 기본적으로 새치기발생에 의해 활성화되기때문에 지연함수가 동작중일 때 례외가 발생할수 없다. 그러므로 국부새치기금지와 스핀잠그기를 함께 쓰는것으로 충분하다.

례외조종기는 local_bh_disable()마크로를 사용하여 간단히 지연함수를 금지할수 있다.(《프로그람적인 새치기》참고) 지연함수를 금지하면 해당 CPU에서 새치기를 계속 처리할수 있으므로 새치기금지보다 더 합리적이다. 매 CPU는 지연함수를 직렬로 실행하므로 경쟁상태는 없다. 마찬가지로 다중처리기체계에서는 항상 하나의 핵심부조종경로만 자료구조체를 호출할수 있도록 하기 의한 스핀잠그기가 필요하다.

○ 새치기와 지연함수에서 호출하는 자료구조체보호

이 경우는 새치기와 례외조종기가 호출하는 자료구조체의 경우와 비슷하다.

지연함수를 실행중일 때 새치기가 발생할수 있지만 지연함수는 새치기조종기를 멈출수 없다.

따라서 국부새치기를 금지하여 경쟁상태를 막아야 한다. 그렇지만 새치기조종기는 다른 새치기조종기가 같은 자료구조체를 호출하지 않는다면 지연함수가 호출하는 자료구조체를 새치기를 금지하지 않고도 자유롭게 다룰수 있다. 마찬가지로 다중처리기체계에서는 여러개의 CPU에서 동시에 자료구조체를 호출하는것을 금지하기 위해 항상 스핀잠그기가 필요하다.

○ 례외와 새치기, 지연함수에서 호출하는 자료구조체보호

우의 경우와 마찬가지로 경쟁상태를 피하기 위해 국부새치기금지와 스핀잠그기가 항상 필요하다. 새치기조종기의 실행을 마칠 때 지연함수를 실행하므로 직접적으로 지연함수를 금지할 필요는 없다. 따라서 국부새치기금지로도 충분하다.

2) 경쟁상태방지실례

핵심부개발자는 겹치는 핵심부조종경로에 의해 발생하는 동기화문제를 파악하고 해결해야 한다. 그렇지만 경쟁상태를 피하려면 핵심부의 여러 구성요소들이 어떻게 호상작용하는가를 정확히 리해해야 하므로 매우 어려운 작업이다. 실지 핵심부코드안에 무엇이 있는가를 알수 있도록 앞에서 본 동기화기법을 사용하는 몇가지 전형적인 실례를 보여준다.

○ **참조계수기**

핵심부내부에서 동시에 자원을 할당하거나 해제함으로써 발생할수 있는 경쟁상태를 막기 위해 참조계수기를 광범하게 사용한다. 《참조계수기(reference counter)》는 기억기페지나 모듈, 파일과 같은 특정한 자원에 관련된 atomic_t계수기이다.

핵심부조종경로에서 자원을 사용하기 시작하면 이 계수기를 하나 증가시키고 자원사용을 끝마치면 감소시킨다. 참조계수기가 0이 되면 자원을 사용하지 않는것으로 보고 필요하다면 자원을 해제할수 있다.

○ **대역 핵심부잠그기**

이전판본의 Linux핵심부에는《대역핵심부잠그기(global kernel lock, 대형핵심부잠그기(big kernel lock) 줄여서 BKL이라고도 한다.)》를 광범하게 사용하였다.

Linux핵심부 2.0판에서 이 잠그기는 동시에 한 처리기만 핵심부방식에서 실행할수 있도록 하는 스핀잠그기였다. Linux핵심부 2.2는 훨씬 유연해져서 더는 한개의 스핀잠그기에 의존하지 않고 많은 수의 핵심부자료구조체를 특정화된 스핀잠그기로 보호하였다. 한편 대형잠그기를 여러개의 작은 잠그기들로 분할하여 교착과 경쟁상태를 막는 일은 그리 간단하지 않아서 대역핵심부잠그기를 남겨두었다. 그래서 아직까지 서로 관련이 없는 여러 핵심부코드를 대역핵심부잠그기를 사용하여 직렬화하였다. Linux핵심부 2.4에서도 대역핵심부잠그기의 역할을 크게 축소하였다. 현재 안정판본에서 대역핵심부잠그기는 대부분 가상파일체계(Virtual File System)에 대한 호출을 직렬화하거나 핵심부모듈을 적재하거나 해제할 때 발생할수 있는 경쟁상태를 막기 위해 사용한다. 이전 안정판본과 비교하면 크게 발전한 점은 망전송이나 파일호출(정규파일에 읽거나 쓰기)을 대역핵심부잠그기로 직렬화하지 않는다는것이다.

Linux핵심부 2.4이후판본에서는 대역핵심부잠그기가 완전히 없어졌다.

○ **기억기서술자읽기/쓰기신호기**

mm_struct자료구조체형의 매 기억기서술자는 mmap_sem마당에 자기만의 신호기를 가지고있다.(《기억기서술자》 sched.h 참고)

이 신호기는 여러개의 프로쎄스에서 한개의 기억기서술자를 공유할수 있기때문에 발생할수 있는 경쟁상태를 방지한다. 례를 들어 핵심부가 프로쎄스에 새로운 기억기구역을 할당하거나 이미 할당한 구역을 확장하려고 하는 경우를 보자.

핵심부가 do_mmap()함수를 호출하면 이 함수는 새로운 vm_area_struct자료구조체를 할당한다.

이 과정에 사용가능한 기억기가 없으면 현재프로쎄스를 보류하고 같은 기억기서술자를 공유하는 다른 프로쎄스를 실행할수 있다. 신호기가 없다면 두번째 프로쎄스가 기억기서술자를 호출하는 어떤 작업이라도 하면(례를 들면 쓰기복사로 인한 페지절환) 자료구조체를 파괴할수 있다. 페지절환례외조종기(《페지절환례외조종기》참고)같은 몇가지 핵심부함수는 기억기서술자를 읽기만 하기때문에 이 신호기는 읽기/쓰기신호기로 실현

한다.

○ **스랩캐쉬목록신호기**

스랩캐쉬(slab cache)서술자의 목록(《캐쉬서술자》참고)을 cache_chain_sem신호기로 보호한다. 이 신호기는 목록을 호출하고 수정하는 배타적인 권한을 부여한다.(\mm\slab.c에서 선언한 semaphore구조체형의 정적변수)

kmem_cache_shrink()함수나 kmem_cache_reap()함수가 목록을 차례로 검색할 때 kmem_cache_create()함수가 목록에 새로운 항목을 추가하는 경우 경쟁상태가 발생할수 있다.

```
int kmem_cache_shrink(kmem_cache_t *cachep)
{
   if (!cachep || in_interrupt())
         BUG();

   return __cache_shrink(cachep);
}
```

그렇지만 새치기를 처리하는 도중에는 이 함수를 호출하지 않으며 이 함수는 목록을 호출하고있는 동안에는 프로쎄스를 차단하지 않는다.

핵심부가 비선취형이므로 이 함수는 단일처리기체계에서 겹치지 않는다. 그렇지만 이 신호기는 다중처리기체계에서 중요한 역할을 수행한다.

○ **색인마디신호기**

Linux는 디스크파일에 대한 정보를 《색인마디(inode)》라는 기억기객체에 보관한다.

해당 자료구조체에는 자기만의 신호기인 i_sem마당이 있다. 파일체계를 다루는 과정에서는 경쟁상태가 많이 발생할수 있다. 실제로 디스크에 있는 매 파일은 모든 사용자가 사용할수 있는 자원이다.

모든 프로쎄스는 (잠재적으로)파일내용을 호출하거나 파일이름이나 경로를 바꾸고 파일을 지우거나 복제하는 작업을 할수 있다. 례를 들어 한 프로쎄스가 어떤 등록부에 있는 파일목록을 본다고 하자. 매 디스크작업은 잠재적으로 프로쎄스를 차단할수 있으며 따라서 단일처리기체계라고 하더라도 다른 프로쎄스가 똑같은 등록부에 호출하여 이전 프로쎄스가 등록부목록을 보는중인데도 등록부의 내용을 변경시킬수 있다. 또는 서로 다른 프로쎄스가 동시에 같은 등록부를 수정할수 있다. 이런 모든 경쟁상태를 색인마디신호기를 리용하여 등록부를 보호함으로써 막을수 있다. 프로그람이 두개 이상의 신호기를 사용할 때마다 서로 다른 두개의 조종경로에서 서로 상대방이 신호기를 해제하길 기다리는 교착(deadlock)상태가 발생할 가능성이 있다. 일반적으로 Linux에서 매 핵심부조종경로는 대부분이 한번에 신호기를 단 하나만 획득해야 하므로 신호기요구와 관련한 교착문제는 거의 없다. 그렇지만 핵심부가 두개의 신호기잠그기를 얻어야 하는 경우가 몇

가지 있다. 바로 색인마디신호기에서 이런 상태가 발생하기 쉽다.

례를 들면 rmdir()와 rename()체계호출을 처리하는 봉사루틴에서 이런 상황이 발생한다.(두 경우에 그 작업에 두개의 색인마디가 관련되기때문에 두개의 신호기를 모두 획득해야 한다.) 이러한 교착을 피하기 위하여 주소순서에 따라 신호기에 대한 요구를 수행한다. 즉 semaphore자료구조체가 있는 주소가 낮은 신호기에 대한 요구를 먼저 하는것이다.

제 2 절. 시간동기측정

콤퓨터에서 이루어지는 수많은 동작은 시간동기측정(timing measurement)에 따라 동작하며 이것은 때때로 사용자도 모르게 이루어진다. 례를 들어 콤퓨터콘솔을 일정한 시간동안 사용하지 않을 때 자동으로 화면이 꺼진다면 이것은 핵심부에 사용자가 건반을 누르거나 마우스를 움직인후 시간이 얼마나 지났는가를 알수 있게 하는 시계가 있기때문이다. 체계가 사용하지 않는 파일을 지울것인가를 묻는다면 이것은 모든 사용자파일중에서 오랜 시간동안 사용하지 않은 파일을 구별할수 있는 프로그람이 알려주기때문이다. 이런일을 하려면 프로그람은 파일에 마지막으로 호출한 시간의 시간정보(timestamp)를 알아낼수 있어야 하며 핵심부가 이러한 시간정보를 자동으로 기록해야 한다. 더 중요한 작업으로 시간동기를 통해 프로세스를 절환하는것뿐만 아니라 시간넘침(time-out)을 검사하는것과 같은 눈으로 볼수 있는 핵심부작업을 할수 있다.

Linux핵심부가 수행해야 하는 시간동기측정은 크게 두가지 종류로 나눌수 있다.

· 현재 날자와 시간을 유지하여 사용자프로그람이 time()이나 ftime(), gettimeofday() 체계호출(《time(), ftime(), gettimeofday()체계호출》을 참고)을 호출하면 이것을 되돌리고 핵심부자체에서 파일과 망파케트(packet)의 시간정보로 사용한다.

· 핵심부나(《시계의 역할》참고) 사용자프로그람에(《setitimer()와 alarm()체계호출》 참고) 어떤 정해준 시간이 지났다는것을 알려주는 기구인 시계를 관리한다.

시간동기측정은 고정된 주파수로 동작하는 발진기(oscillator)와 계수기(counter)를 기반으로 하는 여러가지 하드웨어회로를 통해 수행한다.

이 절은 크게 네 부분으로 이루어진다. 먼저 시간동기의 토대가 되는 하드웨어장치를 설명하고 Linux에서 시간을 관리하는 전체적인 구조를 보여준다. 다음은 CPU시분할(time sharing)실현이나 체계시간과 자원사용통계갱신, 쏘프트웨어시계관리 등 핵심부가 해야 하는 시간과 관련된 주요임무를 설명한다.

마지막에 시간동기측정과 관련한 체계호출과 해당 봉사루틴을 설명한다.

1. 하드웨어시계

80x86기본방식에서 핵심부는 실시간시계(RTC: Real Time Clock), 시간정보계수기(TSC: Time Stamp Counter), 프로그람가능한 간격시계(PIT: Programmable Interval Timer) 그리고 SMP 체계에 있는 국부APIC시계라는 4가지 시계와 호상작용해야 한다. 앞에 있는 두가지 장치는 핵심부가 현재 날자와 시간을 알수 있게 한다. 핵심부는 PIC장치와 국부APIC시계를 프로그람적으로 미리 정해놓은 고정된 주파수로 새치기가 발생할수 있게 한다. 이런 정기적인 새치기는 핵심부와 사용자프로그람에서 사용하는 시계를 실현하는데 필수적이다.

1) 실시간시계

모든 개인용콤퓨터에는 CPU를 비롯한 다른 모든 소편과 무관계하게 동작하는 《실시간시계(RTC:Real Time Clock)》가 있다.

RTC는 콤퓨터의 전원이 꺼진 후에도 작은 충전지나 축전지를 통해 전지를 공급받기때문에 계속 동작한다. CMOS RAM과 RTC는 Motorola 146818이나 이와 동등한 일을 하는 한개의 소편에 통합되여있다. RTC는 2㎐부터 8192㎐사이의 주파수로 IRQ8로 정기적인 새치기를 발생시킬수 있다. 또한 RTC가 특정한 값에 도달하면 IRQ8에 새치기를 발생시키도록 프로그람적으로 지정해놓으면 경보(alarm)시계로도 동작할수 있다.

Linux는 RTC를 날자와 시간을 알아내는 용도로만 사용한다. 그러나 /dev/rtc장치파일을 통해 프로쎄스가 RTC를 프로그람적으로 조종할수 있도록 한다. 핵심부는 0x70과 0x71입출력포구(I/O Port)를 통해 RTC를 호출한다. 체계관리자는 이 두개의 입출력포구에 직접 동작하는 《clock》Unix체계프로그람을 실행하여 시간을 설정할수 있다.

2) 시간정보계수기

모든 80x86극소형처리기는 외부발진기에서 박자(clock)신호를 받는 CLK입력단자를 가진다.

펜티움부터 시작하여 최근에 나온 여러가지 80x86극소형처리기에는 64bit시간정보계수기(TSC: Time Stamp Counter)등록기가 있다. 이 값은 rdtsc기호언어 명령어로 읽을수 있다. 이 등록기는 박자신호를 받을 때마다 증가하는 계수기이다. 례를 들어 박자가 400㎒로 동작하면 시간정보계수기값은 2.5ns마다 증가한다.

Linux는 이 등록기를 리용하여 프로그람가능한 간격시계보다 훨씬 정밀하게 시간을 측정한다. 이를 위해 Linux는 체계를 초기화할 때 박자신호의 주파수를 알아내야 한다. 핵심부를 콤파일할 때 이 주파수를 선언하지 않기때문에 똑같은 핵심부를 박자가 다른 CPU에서 실행할수 있다.

실지 CPU의 주파수를 알아내는 작업은 체계를 기동할 때 진행한다. calibrate_tsc()함수는 어느 정도 긴시간간격(50.00077ms)동안 발생한 박자신호수를 계수하여 주파수를 계산한다.(common.c 참고)

앞에 있는 시간간격상수는 프로그람가능한 간격시계의 통로가운데서 하나를 적당히 설정하여 나온것이다. calibrate_tsc()함수는 체계초기화시에만 호출하므로 실행시간이 길더라도 문제가 되지 않는다.

3) 프로그람가능한 간격시계

실시간시계와 시간정보계수기외에도 IBM호환콤퓨터에는 《프로그람가능한 간격시계(PIT: Programmable Interval Timer)》라는 시간측정장치가 있다.

PIT의 역할은 전자료리기에서 사용자가 료리시간이 지났는가를 알수 있게 해주는 경보시계와 비슷하다. PIT는 종을 울리는 대신 《시계새치기(timer interrupt)》라는 특별한 새치기를 발생시켜 핵심부에 시간간격이 하나이상 지났다는것을 알려준다. 경보시계와 또 다른 차이는 PIT는 핵심부가 지정한 고정된 주파수로 영원히 새치기를 발생시킨다는것이다. 매 IBM호환콤퓨터마다 적어도 PIT가 하나씩 있으며 보통 0x40-0x43 입출력포구를 사용하는 8254CMOS소편을 리용한다.

뒤에서 보지만 Linux는 콤퓨터의 첫번째 PIT를 (대략)100㎐주파수로 즉 10ms마다 한번씩 IRQ0에 시계새치기를 발생시키도록 한다. 이 시간간격을 가리켜 《틱크(tick)》라고 하며 tick변수에 틱크의 길이를 ㎲단위로 변환하여 보관한다. 틱크는 체계의 모든 동작에 박자를 맞추어 준다. 어떤 면에서는 음악가가 련습할 때 박자기(metronom)가 내는 틱크와 비슷하다.

일반적으로 틱크가 짧으면 시계의 해상도가 높아져 다매체재생을 부드럽게 하고 동기적으로 다중입출력을 할 때 (poll()이나 select()체계호출)반응시간이 짧아진다. 그렇지만 틱크가 짧아지면 CPU가 핵심부방식에서 더 많은 시간을 보내야 한다. 즉 사용자방식에 있는 시간이 줄어들고 결과적으로 사용자프로그람의 실행이 느려진다. 따라서 매우 성능이 높은 콤퓨터만이 매우 짧은 틱크을 선택해야 한다. 현재는 HP의 Alpha와 Intel의 IA-64로 실현(porting)한 Linux핵심부만이 초당 1024번 즉 대략 1ms마다 한번씩 시계새치기를 발생시킨다. Alpha station은 가장 높은 틱크주파수를 선택하여 시계새치기를 초당 1200번 발생시킨다.

Linux코드에서는 다음과 같은 마크로들을 통해 시계새치기의 주파수를 결정하는 상수를 만든다.

- HZ는 초당 시계새치기의 회수 즉 시계새치기의 주파수를 나타낸다. IBM PC와 대부분의 다른 하드웨어장치에서 이 값은 100이다.
- CLOCK_TICK_RATE는 8254소편의 내부발진기주파수인 1193182이다.
- LATCH는 CLOCK_TICK_RATE와 ㎐사이의 비률이다. PIT를 프로그람적으로 작성할 때 이것을 사용한다.

setup_pit_timer()함수는 PIT를 다음과 같이 초기화한다.

```
    spin_lock_irqsave(&i8253_lock, flags);
    outb_p(0x34,0x43);
    udelay(10);
    outb_p(LATCH & 0xff, 0x40);
    udelay(10);
    outb
(LATCH >> 8, 0x40);
    spin_unlock_irqrestore(&i8253_lock, flags);
```

C함수 outb()는 기호언어명령어와 같다. 이 함수는 두번째 피연산자로 지정한 입출력포구에 첫번째 피연자로 지정한 값을 복사한다. outb_p()함수는 outb()와 비슷하지만 아무 일도 하지 않는 명령어를 실행하여 잠간 쉰다는 점이 다르다. 맨 처음 호출하는 outb_p()는 PIT가 새로운 속도로 새치기를 발생시키도록 명령한다. 다음에 호출하는 outb_p()와 outb()는 장치에 새로운 새치기속도를 지정한다.

장치의 8bit 0x40입출력포구로 16bit LATCH 상수를 련속 2B를 쓴다. 결과적으로 PIT는 (대략)100㎐ 주파수로(즉 10ms마다) 시계새치기를 발생시킨다.

4) CPU국부시계

최근 Intel처리기에 있는 국부 APTC는 (《새치기와 례외》참고) 또 다른 시간측정장치인 《CPU국부시계(CPU local timer)》기능도 제공한다. CPU국부시계는 방금전에 설명한 프로그람가능한 간격시계와 비슷하게 한번 또는 정기적으로 새치기를 발생시킬수 있는 장치이다. 그렇지만 다음과 같은 몇가지 차이가 있다.

・APIC시계계수기는 32bit크기지만 PIT의 시계계수기는 16bit크기이다. 따라서 매우 낮은 주파수로 새치기를 발생시키도록 국부시계를 프로그람적으로 설정할수 있다.(계수기는 새치기가 발생하기까지 경과하는 틱크수를 보관한다.)

・국부APIC시계는 자기의 처리기에만 새치기를 전달하지만 PIT는 체계에 있는 모든 CPU에서 처리할수 있는 대역새치기를 발생시킨다.

・APIC시계는 모선박자신호(또는 이전장치에서는 APIC모선신호)를 기반으로 한다. 그리고 1번 또는 2, 4, 8, 16, 32, 64, 128번 모선박자신호마다 시계계수기를 줄이도록 프로그람적으로 설정할수 있다. 반대로 PIT는 자기만의 내부박자발진기를 가지고있다.

5) 고정밀도사건시계(HPET)

고정밀도사건시계(HPET)는 인텔과 마이크로쏘프트가 공동으로 개발한 새로운 시계단위이다. HPET를 말단사용자들이 잘 모르지만 Linux 2.6에서는 지원하므로 그 특성에 대하여 몇가지만 서술한다.

HPET는 핵심부가 관리할수 있는 일정한 수의 하드웨어시계를 제공한다. 기초적으로 소편은 8개의 32bit, 혹은 64bit의 독립적인 계수기들을 포함한다. 매 계수기들은 자체의

박자신호에 의해 구동하며 적어도 10MHz이상이므로 계수기는 100ns에 적어도 1번이상 증가된다. 매 계수기는 대체로 32개시계와 련관을 가지는데 비교기와 정합등록기로 구성된다.

HPET소편은 기억기공간속에 넘겨진 등록기를 통해 프로그람화할수 있다.(I/O APIC와 매우 류사하다.)

6) ACPI전원관리시계

이것은 APIC에 기초한 거의 모든 주기판들에 포함되여있는 또 하나의 박자장치이다. 주파수는 대략적으로 3.58MHz이다. 장치는 박자당 단일계수기를 증가한다. 계수기의 현재값을 읽기 위해 핵심부는 초기화시에 BIOS가 결정하는 I/O포구주소에 접근한다.

ACPI전원관리시계는 조작체계나 BIOS가 충전기전원을 보호하기 위해 CPU들의 주파수나 전압을 동적으로 떨구려 할 때 TSC보다 오히려 더 나은 감이 있다.

Linux핵심부가 체계의 모든 작업을 이끌어나가기 위해서 이것을 어떻게 리용하는가를 보자.

2. Linux시간관리구조

Linux는 시간과 관련한 여러가지 동작을 수행한다. 례를 들어 핵심부는 정기적으로 다음과 같은 일을 한다.

· 체계가 기동한 때 부터 경과한 시간을 갱신한다.

· 날자와 시간을 갱신한다.

· 모든 CPU에서 현재프로쎄스가 얼마나 오래동안 실행중인가를 확인하고 이 시간이 프로쎄스에 할당된 시간을 넘어서면 프로쎄스를 선취한다 .

· 자원사용통계를 갱신한다.

· 매 프로그람시계(《시계의 역할》참고)에서 지정한 간격이 지났는가를 검사한다.

Linux의 《시간관리구조》는 시간흐름과 관련있는 핵심부자료구조체와 함수의 집합이다. Intel기반다중처리기콤퓨터의 시간관리구조가 단일처리기콤퓨터와는 조금 다르다.

· 단일처리기체계에서 모든 시간관리작업은 프로그람가능한 간격시계가 발생시키는 새치기에 의해 일어난다.

· 다중처리기체계에서 일반적인 작업(프로그람적인 시계처리 등)은 PIT가 발생시키는 새치기에 의해 일어나지만 CPU마다 일어나는 작업(현재 실행중인 프로쎄스의 실행시간을 감시하는 등)은 국부APIC시계가 발생시키는 새치기에 의해 일어난다.

하지만 이 두가지 경우의 경계는 모호하다. 례를 들어 Intel 80486 처리기기반 초기SMP체계에는 국부APIC가 없다. 심지어 현재까지도 오유가 많아서 국부시계를 사용할수 없는 SMP주기판이 있다. 이런 경우에 SMP핵심부는 단일처리기의 시간관리구조에 의거할수밖에 없다. 반면에 최신단일처리기체계에도 국부APIC와 입출력APIC가

있어서 핵심부는 SMP의 시간관리구조를 리용할수 있다. 다른 중요한 경우로 단일처리기콤퓨터에서 SMP를 지원하는 핵심부를 실행하는 경우가 있다. 그렇지만 간단히 설명하기 위해 이런 복합적인 경우는 취급하지 않고 두가지 《순수한》 시간관리구조만 보기로 한다.

Linux의 시간관리구조는 시간정보계수기(TSC)가 있으면 이것을 리용한다. 핵심부는 두가지 기본적인 시간관리기능을 사용한다. 하나는 현재시간을 최신으로 유지하는것이고 다른 하나는 현재 1s내에서 경과한 μs를 세는것이다. 이 μs를 알아내는 방법은 두가지이다. CPU에 시간정보계수기가 있으면 이것을 리용하는것이 더 정확하지만 없는 경우에는 다른 방법을 사용한다.(《time(), ftime(), gettimeofday()체계호출》참고)

1) 시간관리자료구조체

핵심부2.6의 시간관리구조는 많은 자료구조체들을 리용하도록 되여있다. 일반적으로 80x86구성방식에 기초하여 기본적인 변수들을 서술하게 된다.

•timer객체

가능한 시간관리자원들을 같은 방법으로 처리하기 위해 핵심부는 timer객체를 리용하는데 이 객체는 다음의 표에 제시되는 timer이름과 4개 표준메쏘드들로 이루어지는 timer_opts형의 서술자이다.

표 8-10. timer_opts자료구조체의 마당

마당이름	설명
Name	timer원천식별을 위한 문자렬
mark_offset	마지막순간의 정확한 시간에 대한 기록. 시간새치기처리 때에 쓰인다.
get_offset	마지막순간에 측정된 시간을 되돌려준다.
Monotonic_clock	핵심부초기화후부터 ns계수값을 되돌린다.
Delay	《loops》개수만큼 대기(《지연함수》참고)

timer객체의 기본메쏘드들은 mark_offset와 get_offset이다.

mark_offset메쏘드는 시간새치기처리기에 의해 리용되며 정확한 시간을 자기의 자료구조체에 기록한다. get_offset메쏘드는 기록된 값을 리용하여 마지막시간새치기(tick)때로부터 μs단위로 측정된 시간을 계산한다. 이 두 메쏘드를 리용하여 리눅스시간관리구조는 시간분할의 방도를 얻어낸다. 이로 해서 핵심부는 새치기간격보다 더 높은

정확성을 가지고 현재시간을 결정할수 있다. 이 조작을 시간보정(timer interpolation)이라고 부른다.

cur_timer변수는 체계에서 가능한 《제일 좋은》timer자원에 따라 timer객체의 주소를 저장한다. cur_timer는 핵심부가 초기화될 때 리용되는 dummy timer원천에 대응하는 객체인 timer_none을 지적한다. 핵심부초기화기간에 select_timer()함수는 적당한 timer객체의 주소에 cur_timer를 설정한다. 표 8-11는 참고로 80x86방식에서 쓰이는 기본적인 timer객체들을 보여준다. 알수 있는것처럼 select_timer()는 가능하면 HPET를 선택한다. 다시 말해서 가능하면 ACPI전원관리시계 또는 TSC를 선택한다. 마지막에 select_timer()는 상용PIT를 선택한다. 《시간보정》렬은 timer객체의 mark_offset와 get_offset메쏘드들이 리용하는 timer원천들을 보여주며 《지연》렬은 delay메쏘드가 리용하는 timer원천들을 보여준다.

표 8-11. 80x86방식에서의 표준timer객체들

Timer 객체이름	설명	시간보정	지연
Timer_hpet	고정확도사건Timer (HPET)	HPET	HPET
timer_pmtmr	ACPI 전원관리Timer (ACPI PMT)	ACPI PMT	TSC
Timer_tsc	시간표계수기(TSC)	TSC	TSC
Timer_pit	프로그람가능한 간격시계 (PIT)	PIT	Tight loop
Timer_none	일반 dummy timer원천(핵심부초기화시에 리용)	(none)	Tight loop

국부APIC시계는 대응하는 timer객체가 없다. 그 리유는 국부 APIC시계들이 주기적인 새치기발생에만 쓰이고 시간분할(sub-tick)에는 리용되지 않기때문이다.

•jiffies변수

jiffies변수는 체계가 시작한 때로부터 측정된 박자수를 저장하는 계수기이다. 시계새치기가 발생하면 매 순간 하나씩 증가한다. 80x86방식에서는 jiffies가 32bit변수이므로 리눅스봉사기에 대하여 짧은 시간간격으로 50일에 근사시킨다. 핵심부는 time_after,time_after_eq, time_before, time_before_eq마크로들을 리용하여 jiffies의 자리넘침을 정확히 처리한다.

체계가 시작할 때 jiffies는 0으로 초기화된다고 생각할수 있다. 실제로 jiffies는

0xfffb6c20으로 초기화되는데 이것은 30만을 32bit로 표시한 값에 해당한것이다. 그러므로 계수기는 체계기동후에 5min을 넘길것이다. 이것은 jiffies의 자리넘침검사도 제대로 못하는 오유가 있는 핵심부코드를 개발국면에서 될수록 빨리 보여주며 핵심부안정판에서 무시되지 않도록 할 목적에서 진행된다.

일부 경우에 핵심부는 jiffies 의 자리넘침에는 관계없이체계기동후부터 측정한 체계박자의 실제값을 요구하기도 한다. 그러므로 80x86방식에서는 jiffies변수를 jiffies_64로 부르는 64bit계수기의 기본비트보다 작은 32bit에 대한 련결과 동일시한다. 1ms간격으로 jiffies_64변수는 수백년을 기록할수 있으므로 자리넘침이 전혀없이 안전하다는것을 알수 있다.

80x86방식에서 64bit unsigned long long옹근수형으로 직접 jiffies를 선언하지 못할수 있다고 우려할수 있다. 그에 대한 대답은 32bit방식에서 64bit에 대한 접근이 자동적으로 진행될수 없다는것이다. 그러므로 전체 64bit에 대한 모든 읽기조작은 계수기는 그냥 두고 두개의 32bit짜리 반계수기(half-counter)를 읽는것을 보정하는 동기화기술이 요구된다. 결과적으로 모든 64bit읽기조작은 32bit읽기조작보다 신호적으로 떠진다.

get_jiffies_64()함수는 jiffies_64의 값을 읽어 그 결과를 되돌린다.

```
unsigned long long get_jiffies_64(void)
{
    unsigned long seq;
    unsigned long long ret;
    do {
        seq = read_seqbegin(&xtime_lock);
        ret = jiffies_64;
    } while (read_seqretry(&xime_lock, seq));
    return ret;
}
```

64bit읽기조작은 xtime_lock seqlock에 의해 보호된다. 함수는 정확히 다른 핵심부조종경로에 의해 현재 변경되지 않는다는것을 알 때까지 jiffies_64변수를 계속 읽기한다.

거꾸로 jiffies_64변수를 증가하는 제한지역은 write_seqlock(&xtime_lock)와 write_sequnlock(&xtime_lock)에 의해 보호되여야 한다. ++jiffies_64명령 또한 jiffies_64d의 아래절반에 대응하기때문에 32bit jiffies변수를 증가시킨다.

•xtime변수

xtime변수는 현재 시간과 날자를 저장하는데 다음의 두 마당을 가지는 timesepc형의 구조체이다.

tv_sec : 1970년 1월 1일(UTC) 0시부터 측정된 초수를 기록한다.

tv_nsec : 마지막초내에 측정된 ns의 수를 저장한다.(0부터 999999999사이 범위

의 값을 가진다.)

xtime변수는 보통 매 박자마다 변경되는데 초당 대략 1000번이다. 후에 보겠지만 사용자프로그람은 xtime변수로부터 현재 시간과 날자를 얻는다. 핵심부 또한 자주 그것을 참조하는데 례를 들면 timestamps매듭을 변경할 때 리용한다.

xtime_lock seqlock는 xtime변수에 대한 동시접근에 의하여 일어나는 경쟁조건을 피한다. xime_lock는 jiffies_64변수도 보호하기때문에 일반적으로 이 seqlock는 시간관리구조의 여러가지 제한지역정의에 리용된다.

2) 단일처리기체계의 시간관리구조

단일처리기체계에서 시간과 관련한 모든 작업은 프로그람기능한 간격시계가 IRQ0으로 발생시키는 새치기에 의하여 일어난다. 다른 새치기와 마찬가지로 Linux는 이러한 작업의 일부를 새치기가 발생하는 즉시 처리하고 (새치기조종기의 《상반부(top half)》에서) 나머지 작업은 지연실행한다.(새치기조종기의 《하반부(bottom half)》에서)

○ PIT의 새치기봉사루틴

time_init()함수는 핵심부초기화를 할 때 IRQ0의 새치기문를 설정한다.(time.c 참고)

이것을 수행하고 나면 IRQ0의 irqaction 서술자의 handler마당은 timer_interrupt()함수의 주소를 담는다. IRQ0 주요서술자의 flags마당에 SA_INTERRUPT기발을 설정하기때문에 이 함수는 새치기를 금지한 상태에서 실행된다.

이 함수는 다음과 같은 단계를 실행한다.

1. CPU에 TSC등록기가 있으면 mark_offset_tsc()함수를 실행한다.

이 함수는 다음과 같은 동작을 진행한다.

i. rdtsc기호언어명령어를 실행하여 TSC등록기의 아래자리 32bit를 last_tsc_low변수에 보관한다.

ii. 8253소편장치의 내부발진기상태를 읽어 시계새치기가 발생한 때부터 새치기봉사루틴을 실행할 때까지 지연된 시간을 계산한다.

iii. 지연된 시간을 μs단위로 환산하여 delay_at_last_interrupt변수에 보관한다. 후에 《time(), ftime(), gettimeofday()체계호출》에서 보지만 사용자프로쎄스에 정확한 시간을 제공하려고 이 변수를 사용한다.

2. do_timer_interrupt()함수를 호출한다.

모든 80x86모형에서 공통적인 새치기봉사루틴이라고 할수 있는 do_timer_interrupt()함수는 다음과 같은 동작을 수행한다.

1. do_timer()함수를 호출한다. 이 함수는 곧 자세히 설명한다.

2. 핵심부방식에 있을 때 시계새치기가 발생하였다면 x86_do_profile()함수를 호출한다.

3. adjtimex()체계호출을 실행한 경우 660s마다 한번, 즉 11min마다 한번씩 set_rtc_mmss()함수를 호출하여 실시간시계를 정확히 설정한다. 이 기능은 망으로 련

결된 체계들이 시계를 똑같이 맞출수 있게 한다.(《adjtimex()체계호출》참고)

do_timer()함수는 새치기를 금지한 상태에서 실행하므로 가능한 빨리 실행해야 한다.

이런 리유로 이 함수는 단순히 체계를 시작한 때부터 경과한 시간을 나타내는 기본적인 값 하나만을 갱신하고 현재 실행중인 프로쎄스가 자기의 시간을 다 소비했는가를 검사하고 시간정보를 갱신한다.

이 함수는 다음과 같다.

```
void do_timer (struct pt_regs *regs)
{
        jiffies_64++;
        update_proces  s_times  (user_mode(reqs));  /* 단일처리기체계에서만
   */
        update_times();
        }
```

대역변수 jiffies_64는 체계를 시작한 때부터 경과한 틱크의 수를 보관한다. 핵심부를 초기화할 때 이것을 0으로 설정하고 시계새치기가 발생할 때마다 즉 매 틱크마다 1씩 증가시킨다. jiffies_64는 64bit의 부가 아닌 정수이므로 체계시동후 497일정도 지나면 0으로 돌아간다. 그렇지만 핵심부는 자리넘침(overflow)이 발생하더라도 혼동하지 않고 이것을 잘 처리한다. update_process_times()함수는 기본적으로 현재프로쎄스가 얼마나 오래동안 실행중인가를 검사한다. 《CPU의 시분할》에서 설명한다. 마지막으로 update_times()함수를 호출한다. update_times()함수는 체계날자와 시간을 갱신하고 현재체계부하를 계산한다. 이런 작업은 나중에 《날자와 시간갱신》과 《체계통계갱신》에서 설명한다.

3) 다중처리기체계의 시간관리구조

다중처리기체계에서도 프로그람가능한 간격시계가 발생시키는 시계새치기가 여전히 중요한 역할을 한다. 실례로 해당 새치기조종기는 프로그람시계처리나 체계시간을 최신으로 유지하는것은 특정한 CPU에 국한되지 않는 작업을 수행한다.

단일처리기체계에서와 같이 대부분의 급한일은 새치기조종기의 《상반부》에서 실행하고 (《PIT의 새치기봉사루틴》참고) 나머지 작업은 지연실행한다. 그렇지만 SMP용 PIT새치기봉사루틴은 단일처리기용과는 몇가지 차이가 있다.

·timer_interrupt()함수는 xtime_lock 읽기/쓰기스핀잠그기를 쓰기용으로 획득한다.

국부새치기를 금지하더라도 핵심부는 다른 CPU가 xtime과 last_tsc_low, delay_at_last _interrupt대역변수를 동시에 읽거나 쓰지 않도록 보호해야 한다.(《날자와 시간갱신》참고)

·x86_do_profile()함수는 특정한 CPU에 련관된 일을 수행하기때문에 do_timer_interrupt()함수는 이것을 호출하지 않는다.

・update_process_times()함수는 특정한 CPU에 련관된 일을 수행하기때문에 do_timer()함수는 이것을 호출하지 않는다.

체계에 있는 모든 개별CPU와 련관된 시간관리작업은 두가지이다.

・현재프로쎄스가 해당 CPU에서 얼마나 오래동안 실행중인가를 감시

・CPU의 자원사용통계갱신

Linux 핵심부에서는 전체적인 시간관리구조를 간단하게 하려고 이런 작업을 CPU에 내장된 APIC장치가 발생시키는 국부시계새치기의 조종기에서 처리한다. 이렇게 해서 모든 CPU는 자기의 《개인》자료구조체만 호출하기때문에 스핀잠그기를 호출하는 회수가 줄어든다.

○ 시간관리구조의 초기화

핵심부를 초기화할 때 매 APIC에 얼마나 자주 국부시계새치기를 발생시킬수 있는가를 설정한다. setup_boot_APIC_clock()함수는 다음과 같이 모든 CPU의 국부 APIC를 설정하여 새치기를 발생시키게 한다.

```
void __init setup_boot_APIC_clock(void)
{
    printk("Using local APIC timer interrupts.\n");
    using_apic_timer = 1;

    local_irq_disable();
    calibration_result = calibrate_APIC_clock();
    setup_APIC_timer(calibration_result);
    local_irq_enable();
}
```

calibrate_APIC_clock()함수는 시동하는 CPU에서 틱크(10ms) 한번동안에 국부 APIC가 몇번이나 국부시계새치기를 발생시키는가를 계산한다. 그래서 이 정확한 값을 사용하여 국부APIC가 틱크마다 한번씩 국부시계새치기를 발생시키도록 설정한다. 이것은 setup_APIC_timer()함수의 역할이다. 시동하는 CPU에서는 이 함수를 직접 호출하고 다른 CPU에서는 CALL_FUNCTION_VECTOR처리기간새치기(IPI)를 통해 호출한다.(《처리기간새치기》참고)

모든 국부APIC시계는 공통된 모선박자신호를 사용하기때문에 서로 동기적이다. 즉 시동하는 CPU에서 calibrate_APIC_clock()함수로 계산한 값을 체계에 있는 다른 CPU에서도 사용할수 있다. 그렇지만 체계는 모든 국부시계새치기가 정확히 같은 시간에 발생하기를 바라지 않는다. 이것은 스핀잠그기를 기다려야 하기때문에 상당한 성능저

하를 가져올수 있다. 같은 리유로 한 PIT시계새치기조종기를 실행하고있을 때 국부시계새치기조종기를 실행하면 안된다. 따라서setup_APIC_timer()함수는 매 틱크의 간격내에서 국부시계새치기를 분할한다.

그림 8-4에서 그 실례를 보여준다. 4개의 CPU가 있는 다중처리기체계에서 PIT의 시계새치기에 의해 틱크가 시작한다.

PIT시계새치기가 발생한지 2ms후에 CPU0의 국부APIC가 국부시계새치기를 발생시키고 다시 2ms후에 CPU1의 국부APIC가 새치기를 발생시키는 식이다.

CPU3의 국부시계새치기가 발생한 때로부터 2ms후에 PIT는 IRQ0으로 또 다른 시계새치기를 발생시켜 새로운 틱크가 시작한다.

PIT	CPU0	CPU1	CPU2	CPU3	PIT
2ms	2ms	2ms	2ms	2ms	

그림 8-4. 한번의 틱크안에서 국부시계새치기분할하기

setup_APIC_timer()함수는 국부APIC가 LOCAL_TIMER_VECTOR(보통0xef)벡토르를 포함하는 시계새치기를 발생시키게 한다. 나가서 init_IRQ()함수는 LOCAL_TIMER_VEECTOR와 저준위새치기조종기인 apic_timer_interrupt()함수를 련관시킨다.

○ **국부시계새치기조종기**

apic_timer_interrupt() 기호언어 함수는 다음코드와 비슷하다.

```
apic_timer_interrupt:
        pushl   $LOCAL_TIMER_VECTOR-256
                SAVE_ALL
        movl %esp,%eax
        pushl %eax
        call smp_apic_timer_interrupt
        addl $4,%esp
        jmp ret_form_intr
```

보다싶이 저수준조종기는 다른 저수준새치기조종기와 매우 비슷하다. 고수준새치기조종기인 smp_apic_timer_interrupt()는 다음단계를 실행한다.

1. CPU의 론리번호를 알아낸다.(이것을 n이라고 가정하자.)

2. apic_timer_irqs의 n번째 입구점값을 하나 증가시킨다.(《NMI감시기검사》 참고)

3. 국부APIC로 새치기에 대한 응답(acknowledge)을 한다.

4. irq_enter()함수를 호출해서 local_irq_count배렬의 n번째 입구점을 증가시킨다.

5. smp_local_timer_interrupt() 함수를 호출한다.

6. irq_exit()함수를 호출하여 local_irq_count배렬의 n번째 입구점을 감소시킨다.

smp_local_timer_interrupt()함수는 CPU마다 필요한 시간관리작업을 실행한다. 실지 이 함수는 다음과 같은 단계를 수행한다.

1. 핵심부방식에 있을 때 시계새치기가 발생하였다면 x86_do_profile()함수를 호출한다.

2. update_process_times()함수를 호출하여 현재프로쎄스가 얼마나 오래동안 실행중인가를 검사한다.(《CPU의 시분할》 참고)

4) CPU의 시분할

시계새치기는 실행할수 있는 프로쎄스(즉 TASK_RUNNTNG상태에 있는 프로쎄스)는 CPU시분할(time sharing)하는데 필수적이다. 보통 매 프로쎄스에 《시분할량(quantum)》이라는 제한된 시간을 부여하고 프로쎄스가 끝나지 않은 상태에서 시분할량이 완료되면 schedule()함수를 호출해서 새로 실행할 프로쎄스를 선택한다.

프로쎄스서술자의 counter마당은 프로쎄스에 남은 CPU시간의 틱크수를 나타낸다.

시분할량은 항상 틱크의 배수 즉 약 10ms의 배수이다. 틱크가 발생할 때마다 update_process_times()함수는 counter값을 갱신한다. 이 함수는 단일처리기체계에서는 PIT시계새치기조종기가 다중처리기체계에서는 국부시계새치기조종기가 호출한다. 코드는 다음과 같다.

```
if (current->pid) {
        --current->counter;
        if (current->counter <= 0) {
           current->counter = 0;
               current->need_resched = 1;
        }
}
```

이 코드는 먼저 핵심부가 PID가 0인 프로쎄스 즉 실행중인 CPU의 《교환(swapper)》프로쎄스를 실행하고있는가를 확인한다. TASK_RUNNING상태에 있는 다른 프로쎄스가 없을 때 이 프로쎄스를 실행하므로 이 프로쎄스를 시분할하면 안된다.(《프로쎄스구별하기》참고)

counter가 0 보다 작아지면 프로쎄스서술자의 need_reshed마당을 1로 설정한다.

이 경우 사용자방식으로 실행을 다시 하기 전에 schedule()함수를 호출하여 다른 TASK_RUNNING상태의 프로쎄스가 CPU에서 실행을 재개할수 있게 한다.

5) 날자와 시간갱신

사용자프로그람은 struct timeval형 변수 xtime에서 현재 날자와 시간을 알아낸다. 핵심부 또한 색인마디시간정보(《파일서술자와 색인마디》 참고)를 갱신할 때를 비롯하여 자주 이 변수를 참조한다. 구체적으로 보면 xtime.tv_sec는 1970년 1월 1일(UTC)오전부터 경과한 시간을 초로 환산하여 보관하고 xtime.tv_usec는 마지막초부터 경과한 μs를 보관한다.(따라서 이 값은 0에서 999999 사이에 있다.)

핵심부를 초기화하는동안 time_init()함수를 호출하여 날자와 시간을 설정한다. 이 함수는 get_cmos_time()함수를 호출하여 실시간시계(RTC)에서 시간을 읽은 후 xtime을 초기화한다.

한번 이렇게 하면 핵심부가 더는 RIC를 필요로 하지 않는다. 대신 매번 틱크가 발생할 때마다 실행하는 시간계수새치기에 의존한다.

6) 체계통계갱신

핵심부는 시간과 관련한 여러가지 과제중 하나로 다음용도를 위한 자료를 정기적으로 수집해야 한다.

- 실행중인 프로쎄스의 CPU자원제한검사
- 평균체계부하계산
- 핵심부코드동작분석

○ 현행프로쎄스의 CPU자원제한검사

update_process_times()함수는 (단일처리기체계에서는 PIT의 시계새치기조종기, 다중처리기체계에서는 국부시계새치기조종기가 이 함수를 호출한다.) 현재 실행중에 있는 프로쎄스의 서술자주소를 얻는다.(timer.c 참고)

다음으로 update_one_process()를 호출하는데 이 함수는 do_process_times()함수를 호출하여 사용자프로그람이times()체계호출을 통해 알아낼수 있는 통계정보를 보관하는 몇개의 마당을 갱신한다.

```
static void update_one_process(struct task_struct *p, unsigned long user,
                    unsigned long system, int cpu)
{
   do_process_times(p, user, system);
   do_it_virt(p, user);
   do_it_prof(p);
}
```

이때 사용자방식과 핵심부방식에서 사용한 CPU시간을 구별한다. 이 함수는 다음과 같은 동작을 한다.

1.current의 프로쎄스서술자에 있는 utime마당을 갱신한다. 이 마당은 처리기가 사용자방식에서 실행중일 때 발생한 틱크의 수를 보관한다.

2.current의 프로쎄스서술자에 있는 stime마당을 갱신한다. 이 마당은 프로쎄스가

핵심부방식에서 실행중일 때 발생한 틱크의 수를 보관한다.

3.CPU시간제한에 걸리는가를 검사한다. 걸린다면 current에 SIGXCPU와 SIGKILL신호를 보낸다. 《프로쎄스자원제한》에서 매 프로쎄스서술자에 있는 rlim[RLIMIT_CPU].rlim_max마당을 통해 사용제한을 조종하는 방법을 설명하였다.

프로쎄스의 자식프로쎄스가 사용자방식과 핵심부방식에서 사용한 CPU틱크의 개수를 계수하기 위해 프로쎄스서술자에는 추가로 cutime과 cstime이라는 두 마당이 있다. 효률성을 위해 이 마당은 update_one_process()에서 갱신하지 않고 부모프로쎄스가 자식프로쎄스중 하나의 상태를 물어볼 때 갱신한다.(《프로쎄스끝내기》참고)

○ **체계부하파악**

모든 Unix체계는 체계에서 얼마나 많은 CPU동작이 이루어지는가를 계속 파악한다. 이런 통계정보는 top을 비롯한 여러 관리프로그람에서 사용한다. 사용자는 uptime명령을 내려서 마지막 1min과 5min, 15min동안의 《평균부하(load average)》통계를 볼수 있다.

단일처리기체계에서는 이 값이 0이면 실행할수 있는 활성화된 프로쎄스가 없다는것을 의미하고(교환프로쎄스0을 제외하고) 1이면 한 프로쎄스가 CPU를 100% 사용하고있다는것을, 1보다 큰 값이면 여러개의 활성화된 프로쎄스가 CPU를 공유하고있다는것을 의미한다.(timer.c 참고)

update_times()가 호출하는 calc_load()함수는 체계부하에 대한 정보를 수집한다. calc_load()함수는 TASK_RUNNING이나 TASK_UNINTERRUPTIBLE상태에 있는 프로쎄스의 개수를 계수해서 이 값을 CPU사용통계를 갱신하는데 리용한다.

○ **핵심부코드동작분석**

Linux는 핵심부가 핵심부방식에서 자기의 어느 부분을 실행하는데 시간을 소비하는가를 Linux개발자가 알아낼수 있도록 최소한의 코드동작분석기(profiler)를 제공한다. 이 동작분석기는 핵심부의 《활발한 지점(hot spot)》즉 핵심부코드에서 가장 자주 실행되는곳을 찾아낸다. 핵심부의 활발한 지점을 찾아내면 후에 최적화해야 하는 핵심부함수가 무엇인가를 알수 있으므로 매우 중요하다. 동작분석기는 매우 간단한 몬데카를로알고리듬(Monte Carlo algorithm)에 기초한다. 매번 시계새치기가 발생할 때마다 핵심부는 핵심부방식에 있을 때 새치기가 발생하였는가를 검사한다. 그렇다면 탄창에서 새치기가 발생하기 전의 eip등록기값을 읽어서 새치기가 발생하기 전에 핵심부가 무슨 일을 하고있었는가를 알아낸다. 오랜 시간동안 실행하면 이 표본은 활발한 지점으로 모아진다.

x86_do_profile()함수는 코드동작분석기를 위한 자료를 수집한다. 단일처리기체계에서는 do_timer_interrupt()함수(PIT의 시계새치기조종기), 다중처리기체계에서는 smp_local_timer_interrupt()함수(국부시계새치기조종기)가 이것을 실행한다.

inline void smp_local_timer_interrupt(struct pt_regs * regs)

```
{
   int cpu = smp_processor_id();

   profile_tick(CPU_PROFILING, regs);
   if (--per_cpu(prof_counter, cpu) <= 0) {
          /*
   * 이전에 이 시점에서 /proc/profile파일에 사용자쓰기결
   * 과를 설정하였기때문에 다중처리기는 변화될수 있다.
   * 이 경우에 그에 따르는 APIC timer조절이 필요하다.
           * 이 시점에서 새치기들은 이미 마스크해제된다.
           */
          per_cpu(prof_counter, cpu) = per_cpu(prof_multiplier, cpu);
          if (per_cpu(prof_counter, cpu) !=
                              per_cpu(prof_old_multiplier, cpu)) {
                 __setup_APIC_LVTT(
                              calibration_result/
                              per_cpu(prof_counter, cpu));
                 per_cpu(prof_old_multiplier, cpu) =
                                     per_cpu(prof_counter, cpu);
          }

#ifdef CONFIG_SMP
          update_process_times(user_mode(regs));
#endif
   }

   /*
    * 경로되돌이값이 《길》면 거기에서 매 부분체계는
    * 적당히 잠그기한다. (핵심부잠그기/새치기잠그기).
    *
    * 《긴 경로》에 의한 프로파일작성을 줄이고싶을 때 기호
    * 언어에서 전반적으로 프로파일을 작성한다.
    *
    * 현재 너무 많다. (performance wise),
    * 100MHz P5상에서 초당 10만개 국부새치기를 처리할수 있다.
```

```
        */
    }
```

코드동작분석기를 사용하려면 핵심부를 기동할 때 profile=N(여기서 2^N은 분석할 코드토막의 크기를 나타낸다.)문자렬을 파라메터로 설정해야 한다. 수집한 자료는 /proc/profile함수에서 볼수 있다.

이 파일에 쓰기를 해서 계수기를 0으로 만들수 있다. 다중처리기체계에서는 이 파일에 쓰기를 해서 표본을 추출하는 빈도를 바꿀수 있다.(《다중처리기체계의 시간관리구조》 참고) 그렇지만 보통 핵심부개발자는 /proc/profile파일을 직접 호출하는 대신 readprofile체계명령을 사용한다.

7) NMI감시기검사

다중처리기체계에서는 핵심부개발자를 위해 《감시기체계(watchdog system)》라는 또 다른 기능을 제공한다. 이것은 체계를 멈추게 만드는 핵심부오유를 알아내는데 매우 유용하다.

이런 감시기를 사용하려면 핵심부를 기동할 때 nmi_watchdog파라메터를 설정해야 한다.(nmi.c 참고)

이 감시기는 다중처리기주기판의 하드웨어특징에 기초한다. 여기서는 PIT의 새치기시계를 모든 CPU의 NMI새치기로 전달할수 있게 한다.

cli기호언어명령어로 NMI새치기를 금지할수 없으므로 새치기를 금지한 경우라도 감시기는 교착(deadlock)상태에 들어갈수 있다. 그 결과 틱크가 발생할 때마다 모든 CPU는 무슨 일을 하고있는가에 상관없이 NMI새치기 조종기를 실행하고 이 조종기는 do_nmi()를 호출한다. 이 함수는 CPU의 론리번호 n을 알아내서 apic_timer_irqs배렬의 n번째 입구점을 검사한다. CPU가 제대로 동작하고있다면 이 값은 이전 NMI새치기가 발생했을 때 읽은 결과값과 달라야 한다.

CPU가 제대로 동작하고있다면 apic_timer_irqs배렬의 n번째 입구점은 국부시계새치기조종기에 의해 증가한다.(《국부시계새치기조종기》참고) 이 계수기가 그대로라면 국부시계새치기조종기가 완전히 실행되지 않았다는것을 의미한다.

NMI새치기조종기는 CPU가 멈춘것을 발견하면 모든 경보신호를 낸다. 체계일지파일에 오유통보문를 기록하고 CPU등록기와 핵심부탄창의 내용을 복사하고(핵심부편위(kernel oops)) 마지막으로 현재프로쎄스를 소멸한다. 이것은 핵심부개발자가 무엇이 잘못되였는가를 알수 있게 한다.

8) 프로그람적인 시계

《시계(timer)》는 주어진 시간간격이 지난 다음에 함수를 호출하도록 하는 프로그람적인 기법이다. 《시간넘침(time_out)》은 시계에 지정한 시간간격이 지난 순간을 가리킨다.

시계는 핵심부과 프로쎄스가 모두 광범하게 사용한다. 대부분의 장치구동프로그람은

례외적인 상황을 알아내기 위해 시계를 리용한다. 례를 들어 유연성디스크구동프로그람은 유연성디스크를 한참동안 호출하지 않으면 장치의 전동기를 끄기 위해 시계를 사용하고 병렬인쇄기구동프로그람은 인쇄기의 오유상황을 검출하는데 사용한다.

프로그람작성자는 특정한 함수를 일정한 시간이 지난 후에 실행하도록 할 때도 시계를 많이 사용한다.(《setitimer()와 alarm()체계호출》 참고)

시계를 실현하는것은 상대적으로 쉽다. 매 시계에는 시계가 완료되는 시점을 나타내는 마당이 있다. 처음에 현재 jiffies값에 원하는 틱크의 수를 더해서 이 마당을 계산하며 이 마당값은 변경하지 않는다.

핵심부는 시계를 검사할 때마다 시계의 완료시간마당을 현재 jiffies값과 비교하며 jiffies값이 완료시간보다 크거나 같아지면 시계가 완료된다.

time_after와 time_before, time_after_eq, time_before_eq마크로를 리용하여 이런 비교를 할수 있으며 이 마크로는 jiffies값에서 발생할수 있는 자리넘침(overflow)에도 주의한다.

Linux에는 《동적시계(dynamic timer)》와 《간격시계(interval timer)》라는 두가지 시계가 있다. 동적시계는 핵심부가 사용하고 사용자방식에 있는 프로쎄스는 간격시계를 만들수 있다.

Linux시계와 관련해서 주의할 점이 있다. 항상 활성화된 후 오랜시간뒤에 실행해도 문제가 없는 지연함수에서 시계함수를 검사하기때문에 핵심부는 시계함수를 완료된 시간에 정확히 실행한다는것을 보장하지 않는다. 핵심부는 단지 시계함수를 정확한 시간이나 그 시간에서 많으면 몇백ms안에 실행한다는것을 보장할뿐이다. 이런 리유로 시계는 완료시간을 정확하게 지켜야 하는 실시간응용프로그람에는 적당하지 않다.

○ **동적시계**

《동적시계(dynamic timer)》는 동적으로 만들고 없앨수 있다. 현재 사용할수 있는 동적시계개수에는 제한이 없다. 매 동적시계는 다음 timer_list구조체에 보관한다.

```
struct timer_list {
    struct list_head entry;
    unsigned long expires;

    spinlock_t lock;
    unsigned long magic;

    void (*function)(unsigned long);
    unsigned long data;

    struct tvec_t_base_s *base;
```

};

function마당은 시계가 완료될 때 실행할 함수의 주소를 보관한다. data마당은 이 시계함수에 전달할 파라메터를 지정한다. data마당에 일반적인 함수를 하나 정의하여 여러 장치구동프로그람의 시간넘침을 처리할수 있다. data마당에 장치의 ID나 함수에서 장치를 구별하는데 사용할수 있는 다른 의미있는 자료를 보관할수 있다.

expires마당은 시계가 완료될 시점을 지정한다. 이 시간은 체계를 시작한 때부터 경과한 틱크의 수로 나타낸다. expires값이 jiffies값보다 작거나 같은 모든 시계를 완료됐거나 오유가 발생한것으로 간주한다.

entry마당은 사슬형2중련결목록에 대한 련결를 보관한다. 동적시계를 리용하는 사슬형2중련결목록은 512개가 있어서 매 시계를 expires마당의 값에 따라 이 목록가운데서 하나에 삽입한다. 이 목록을 리용하는 알고리듬은 아래에서 설명한다.

핵심부는 다음과 같이 동적시계를 만들고 활성화한다.

1. 새로운 timer_list객체(t라고 하자.)를 만든다. 이것은 여러가지 방법으로 할수 있다.

- 코드에 정적대역변수를 정의한다.
- 함수내에 국부변수를 정의한다.
- 이 객체를 포함하는 서술자를 동적으로 할당한다.

2. init_timer(&t)함수를 호출하여 객체를 초기화한다. 이 함수는 간단히 list마당을 NULL로 설정한다.

3. function마당을 시계가 완료될 때 호출할 함수의 주소로 설정한다. 필요하다면 이 함수에 넘겨줄 파라메터를 data마당에 지정한다.

4. 동적시계를 아직 목록에 삽입하지 않았다면 원하는 값을 expires마당에 지정한다. 이미 목록에 삽입한 경우에는 mod_timer()함수를 호출하여 expires마당을 갱신한다. 이 함수는 객체를 정확한 목록으로 옮기는 일도 한다.

5. 동적시계를 아직 목록에 삽입하지 않았다면 add_timer(&t)함수를 호출해서 t객체를 정확한 목록에 삽입한다. 시계가 완료되면 핵심부는 자동으로 t객체를 목록에서 제거한다. 그러나 때로는 프로쎄스가 del_timer()나 del_timer_sync()함수를 호출하여 시계를 목록에서 제거하기도 한다. 따라서 잠든 프로쎄스는 시간넘침이 되기전에 깨여날수 있으며 이 경우 프로쎄스는 다시 시계를 제거하려고 할수 있다. 이미 목록에서 제거한 시계에 대해 del_timer()나 del_timer_sync()를 호출하여도 아무런 문제가 발생하지 않으므로 시계함수내에서 시계를 제거하는것은 좋은 습관이라고 할수 있다.

○ **동적시계와 경쟁상태**

동적시계는 비동기적으로 활성화되기때문에 경쟁상태를 일으키기 쉽다. 례를 들어 어떤 동적시계함수가 기억기에서 제거할수 있는 자원(례를 들면 핵심모듈이나 파일자료구조체)을 리용하여 동작한다고 하자. 이때 시계를 멈추지 않고 자원을 제거하여 시계함

수가 활성화될 때에는 해당 자원이 존재하지 않는다면 오유가 발생할수 있다. 따라서 가장 중요한 규칙은 《자원을 해제하기 전에 시계를 정지》하는것이다.

```
···
del_timer(&t);
X_Release_Resources();
···
```

그렇지만 다중처리기체계에서는 del_timer()를 호출할 당시 이미 다른 CPU에서 해당 시계함수를 실행하고있을수도 있으므로 이 코드는 안전하지 않다. 이 경우 시계가 해당 자원을 리용하여 동작하는 도중에 자원을 제거해버릴수도 있다. 이런 경쟁상태를 막기 위해서 핵심부는 del_timer_sync()함수를 제공한다. 이 함수는 목록에서 시계를 제거한 후 다른 CPU에서 해당 시계함수를 실행중인가를 검사한다. 그렇다면 del_timer_sync()는 시계함수를 끝마칠 때까지 기다린다.

물론 다른 류형의 경쟁상태도 있다. 례를 들어 이미 활성화한 시계의 expiers마당을 정확한 방법은 mod_timer()를 사용하는것이지만 그렇지 않고 시계를 제거하고 다시 생성한다면 같은 시계의 expires마당을 수정하려는 핵심부조종경로가 두개 있는 경우 서로 혼동할수 있다. 시계함수를 실현할 때에는 timerlist_lock스핀잠그기를 사용하여 SMP에서도 안전하게 만든다. 핵심부는 동적시계목록을 호출할 때 새치기를 금지한 후 이 스핀잠그기를 획득한다.

○ **동적시계처리**

동적시계를 실현하기 위한 적당한 자료구조체를 선택하는것은 쉽지 않다. 모든 시계를 한 목록에 모아둔다면 매번 틱크가 발생할 때마다 긴 시계목록을 검색하기 위하여 상당한 시간을 소비하여 체계성능을 저하시킬것이다. 그렇다고 목록을 정렬해서 유지하는 것 역시 삽입과 삭제에 상당한 시간이 필요하므로 그리 효률적이지 못하다. 따라서 expires값에 기초하여 틱크의 블로크로 나누고 동적시계가 큰 expires 값의 목록으로부터 작은 값의 목록으로 효률적으로 이동하게 하는 자료구조체를 사용하는것이다.

핵심자료구조체는 struct tvec_t_base_s형변수인 base배렬를 리용하는데 매 원소는 tv1, tv2, tv3, tv4, tv5구조체로 된 다섯개의 목록그룹들을 가리킨다. (그림 8-5 참고)

```
typedef struct tvec_s {
   struct list_head vec[TVN_SIZE];
} tvec_t;

typedef struct tvec_root_s {
   struct list_head vec[TVR_SIZE];
} tvec_root_t;
```

```
struct tvec_t_base_s {
    spinlock_t lock;
    unsigned long timer_jiffies;
    struct timer_list *running_timer;
    tvec_root_t tv1;
    tvec_t tv2;
    tvec_t tv3;
    tvec_t tv4;
    tvec_t tv5;
} ____cacheline_aligned_in_smp;

Ty
pedef struct tvec_t_base_s tvec_base_t;
```

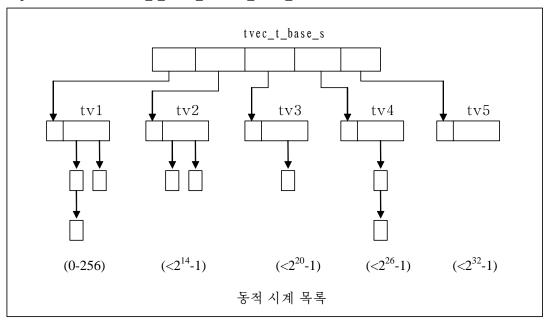



그림 8-5. 동적시계와 련관된 목록의 그룹

tv1구조체는 tvec_root_t형이다. 이 구조체는 list_head객체 256개를 포함하는 vec배렬 즉 동적시계목록으로 이루어진다. 여기에는 틱크가 255번안에 완료되는 모든 동적시계가 들어있다. index로 참조하는 목록에는 현재틱크에서 완료되는 모든 동적시계가 들어있다. 그 다음 목록에는 다음틱크에 완료되는 모든 동적시계가 들어있고 (index+k)번째 목록에는 k번째 틱크가 발생할 때 완료되는 모든 동적시계가 들어있다.

index가 다시 0이 되면 이것은 tvl에 있는 모든 시계를 조사하였다는것을 의미한다. 이 경우 tv2.vec[tv2.index]가 가리키는 목록을 tv1로 공급한다.

tv2, tv3, tv4 구조체는 tvec_t형이며, 각각 $2^{14}-1$, $2^{20}-1$, $2^{26}-1$틱크안에 완료되는 모든 동적시계를 가진다. tvec_t구조체는 tvec_root_t구조체와 매우 비슷하다. 이 구조체는 동적시계목록을 가리키는 64개의 지적자배렬인 vec성원변수를 가지고있다. $256 \times 64^{i-2}$틱크마다 1씩 증가한다.(64로 나눈 나머지값을 보관한다.) 여기서 j는 2와 5사이의 값으로 tvi그룹번호를 의미한다. tv1의 경우처럼 index가 0으로 되돌아오면 tvj.vec[tvj.index]가 가리키는 목록을 tvi(i는 2에서 4사이고 j는 i+1이다.)로 공급한다.

따라서 tv2의 첫번째 요소는 tv1시계 다음 256틱크동안에 완료되는 모든 시계의 목록을 가진다. 따라서 이 목록에 있는 시계를 tv1의 전체 배렬로 공급할수 있다.

tv2의 두번째 요소는 그 다음 256틱크동안에 완료되는 모든 시계를 가지는 방식으로 구성된다. 비슷하게 tv3의 입구점하나로 tv2전체 배렬에 공급할수 있다.

그림 8-5는 이 자료구조체들이 어떻게 서로 련결되는가를 보여준다.

__run_timers()함수는 다음 C코드와 비슷하다.

```
struct timer_list *timer;
   spin_lock_irq(&base->lock);
   while (time_after_eq(jiffies, base->timer_jiffies)) {
          struct list_head work_list = LIST_HEAD_INIT(work_list);
          struct list_head *head = &work_list;
          int index = base->timer_jiffies & TVR_MASK;

          if (!index &&
                 (!cascade(base, &base->tv2, INDEX(0))) &&
                        (!cascade(base, &base->tv3, INDEX(1))) &&
                               !cascade(base, &base->tv4, INDEX(2)))
                 cascade(base, &base->tv5, INDEX(3));
          ++base->timer_jiffies;
          list_splice_init(base->tv1.vec + index, &work_list);
repeat:
          if (!list_empty(head)) {
                 void (*fn)(unsigned long);
                 unsigned long data;

                 timer = list_entry(head->next,struct timer_list,entry);
                 fn = timer->function;
```

```
                data = timer->data;

                list_del(&timer->entry);
                set_running_timer(base, timer);
                smp_wmb();
                timer->base = NULL;
                spin_unlock_irq(&base->lock);
                fn(data);
                spin_lock_irq(&base->lock);
                goto repeat;
            }
        }
        set_running_timer(base, NULL);
        spin_unlock_irq(&base->lock);
```

맨 바깥 while순환은 timer_jiffies가 jiffies값보다 커지면 끝난다. 보통 jiffies값과 timer_jiffies값은 일치하므로 이 while 순환을 한번만 실행하는 경우가 많다. 일반적으로 이 순환을 련속해서 jiffies-timer_jiffies+1번 실행한다. __run_timers()를 실행하는 도중에 시계새치기가 발생하면 jiffies변수는 PIT의 새치기조종기에 의해 비동기적으로 증가하므로([PIT의 새치기봉사루틴]참고) 이번 틱크에서 완료되는 동적시계도 처리한다.

맨 바깥순환을 한번 실행하는 동안 tv1.vec[index]목록에 있는 동적시계함수를 실행한다. 순환에서는 동적시계함수를 실행하기 전에 list_del()함수를 호출하여 동적시계를 목록에서 삭제한다. tv1.index가 비워지면 tv1에 있는 모든 목록을 검사한것이다. 이 경우 tv1구조체를 다시 채워야 하는데 이것은 바로 cascade()함수가 하는 일이다. 이 함수는 다음256틱크동안에 완료되는 tv2.vec[index]에 있는 동적시계를 tv1.vec로 옮긴다. tv2가 비워지면 목록의 tv2배렬을 tv3.vec[index]에 있는 요소로 채운다. 이 함수에서 맨 바깥순환을 시작하기 전에 새치기를 금지하고 base->lock스핀잠그기를 획득한다.

매 동적시계함수를 호출하기 전부터 이 함수가 끝날 때까지 새치기를 허용하고 스핀잠그기를 해제한다. 이렇게 하는것은 다른 핵심부조종경로가 동시에 동적시계자료구조체를 호출하여 자료가 파괴되는것을 막기 위해서이다.

이 알고리듬은 매우 복잡하지만 높은 성능을 보장한다. 그것은 이 함수가 시계새치기 256번중 255번(99.6%)은 완료된 시계가 있는 경우에 이 시계함수를 실행하면 되기때문이다. 그리고 tv1.vec를 정기적으로 채울 때도 64번중에서 63번은 tv2.vec[index]가 가리키는 목록을 tv1.vec의 목록 256개로 나누어넣으면 된다.

tv2.vec배렬은 0.02%의 빈도로(즉 163s에 한번씩) 채우면 된다. 비슷하게 tv3은 2h 54min에 한번씩, tv4는 7일 18h에 한번씩 채우고 tv5는 채울 필요가 없다.

○ **동적시계응용**

핵심부에서 앞에서 나오는 모든 동작을 실제로 어떻게 활용하는가를 보기 위해 《프로쎄스시간넘침(process time-out)》을 만들고 사용하는 실례를 보자.

핵심부가 현재프로쎄스를 2s동안 멈추고 싶다고 하자. 다음코드를 리용하여 할수 있다.

```
timeout = 2 * Hz;
set_current_state(TASK_INTERRUPTIBLE); /* 또는 TASK_UNINTERUP
TIBLE */
remaining = schedule_timeout(timeout);
```

핵심부는 동적시계를 리용하여 프로쎄스시간넘침을 실현한다. 이것은 schedule_timeout()함수에서 볼수 있는데 이 함수는 다음코드와 같다.

```
struct timer_list timer;
expire = timeout + jiffies;
init_timer(&timer);
timer.expires = expire;
timer.date = (unsigned long) current;
timer.function = process_timeout;
add_timer(&timer);
schedule(); /* 시계가 완료될 때까지 프로쎄스를 보류한다 */
del_timer_sync(&timer);
timeout = expire - jiffies;
return timeout < 0 ? 0 : timeout;
```

schedule()함수를 호출하면 실행할 다른 프로쎄스를 선택한다. 이전프로쎄스가 실행을 재시작하면 우의 함수는 동적시계를 삭제한다. 마지막 코드에서 시간넘침이 일어나면 0, 다른 리유로 프로쎄스가 깨여난 경우에는 시간넘침이 완료될 때까지 남은 틱크의 수를 되돌린다.

시간넘침이 일어나면 핵심부는 다음함수를 실행한다.

```
static void process_timeout(unsigned long __data)
{
    wake_up_process((task_t *)__data);
}
```

__run_timers()함수는 timer객체의 data마당에 보관한 프로쎄스서술자지적자를 파라메터로 하여 process_timeout()함수를 호출한다. 이렇게 하면 정지되여있던 프로쎄

스가 깨여난다.

9) **시간동기측정과 관련된 체계호출**

사용자방식프로쎄스가 날자와 시간을 알아내거나 변경하고 시계를 만들수 있게 하는 여러가지 체계호출이 있다. 이러한 체계호출을 간단히 보고 핵심부가 이것을 어떻게 처리하는가를 설명한다.

○ **time(), gettimeofday()체계호출**

사용자방식에 있는 프로쎄스는 여러가지 체계호출을 리용하여 현재 날자와 시간을 알아낼수 있다.

• time()체계호출

1970년 1월 1일(UTC) 오후부터 시작하여 경과한 시간을 s단위로 돌려준다.

• gettimeofday()

1970년 1월 1일 (UTC) 자정부터 시작하여 지금까지 경과한 s단위시간을 timeval이라는 자료구조에 담아 반환한다. (두번째 파라메터인 timezone자료구조는 현재 사용하지 않는다.)

처음 두 체계호출을 gettimeofday()로 대체할수 있지만 이전판본과 호환성을 유지하기 위해 Linux에 계속 들어있다. 이 두 함수는 더는 언급하지 않는다.

gettimeofday()체계호출은 sys_gettimeofday()함수로 구현한다. 이 함수는 현재 날자와 시간을 계산하기 위해 다음과 같이 동작하는 do_gettimeofday()를 호출한다.

1. 새치기를 금지하고 xtime_lock 읽기/쓰기스핀잠그기를 읽기용으로 획득한다.

2. do_gettimeoffset 변수가 가리키는 함수를 호출하여 마지막s부터 경과한 μs를 알아낸다.

usec = do_gettimeoffset();

CPU에 시간정보계수기가 있으면 do_fast_gettimeoffset()함수를 실행한다. 이 함수는 rdtsc 기호언어명령어를 사용하여 TSC등록기를 읽은 후 여기서 last_tsc_low값을 빼서 마지막시계새치기를 처리한 후부터 경과한 CPU주기의 수를 계산한다. 이 수자를 μs로 변환한 후 시계새치기조종기를 실행하기까지 걸린 지연시간(이 값은 앞서 《PIT의 새치기봉사루틴》에서 언급한 delay_last_interrupt변수에 들어있다.)을 더한다.

CPU에 TSC 등록기가 없으면 do_gettimeoffset은 do_slow_gettimeoffset()함수를 가리킨다. 이 함수는 8254소편장치의 내부발진기상태를 읽어서 마지막시계새치기가 발생한 때부터 경과한 시간을 계산한다. 이 값과 jiffies의 내용으로 마지막s이후에 경과한 μs를 알아낼수 있다.

3. 하반부를 아직 실행하지 않은 모든 시계새치기를 고려하여 μs를 증가시킨다.

usec += (jiffies - wall_jiffies) * (1000000/Hz);

4. xtime의 내용을 체계호출을 실행할 때 전달한 파라메터 tv로 지정한 사용자공간의 완충기로 복사한다.

```
        tv->tv_sec = xtime->tv_sec;
    tv->tv_use = xtime->tv_usec+usec;
```

5. xtime_lock스핀잠그기를 해제하고 새치기를 다시 허용한다.

6. μs마당에 자리넘침(overflow)이 있는지 검사하여 필요하면 μs와 s마당을 조정한다.

```
while (tv->tv_usec >= 1000000) {
        tv->tv_usec -= 1000000;
    tv->tv_sec++;
}
```

뿌리권한이 있는 사용자방식프로쎄스는 낡은 stime()체계호출이나 settimeofday()체계호출을 리용하여 현재 날자와 시간을 바꿀수 있다. sys_settimeofday()함수는 do_gettimeofday()와 반대로 동작하는 do_settimeofday()를 호출한다.

두 체계호출모두 RTC의 등록기를 변경하지 않고 xtime값만 바꾸는데 주의하시오. 체계를 완료하면 새로 지정한 값으르 읽기때문에 그렇지 않으려면 clock프로그람을 실행하여 RTC값을 바꾸어야 한다.

○ adjtimex()체계호출

시간이 흐르면 결국 모든 체계가 정확한 시간에서 벗어나지만 그렇다고 갑자기 시간을 바꾸는것은 관리측면에서 귀찮고도 매우 위험한 행동이다. 례를 들어 make프로그람을 사용하여 파일에 적힌 시간정보를 바탕으로 오래된 오브젝트파일을 다시 콤파일하여 큰 프로그람을 건립(build)하는 경우를 생각해보자. 이때 체계시간을 크게 바꾸면 make프로그람을 혼동시켜 잘못 건립할수도 있다. 콤퓨터망에서 분산파일체계를 구축할 때 시간을 맞추는것역시 중요하다. 여기서는 파일에 호출할 때 해당 파일의 색인마디에 있는 시간정보값이 일치하도록 서로 련결된 PC의 시계를 맞추는것이 현명하다.

따라서 각 티크가 발생할 때마다 점차적으로 시간을 변경하는 원칙을 사용하는 망시간통신규약(NTP: Network Time Protocol) 같은 시간동기화통신규약을 실행하도록 체계를 설정하는 경우가 종종 있다. 이 유틸리티는 Linux에 있는 adjtimex()체계호출을 사용한다.

이 체계호출은 여러 Unix변종에 존재하지만 프로그람에서 호환성을 고려한다면 이것을 사용하면 안된다. 이 체계호출은 timex구조체에 대한 지적자를 파라메터로 받아서 timex에 있는 여러 마당의 값으로 핵심부파라메터를 갱신하며 똑같은 구조체에 현재핵심부의 값을 넣어 반환한다. 이 핵심부값은 update_wall_time_one_tick()함수에서 각 티크마다 xtime.tv_usec에 더하는 μs수를 미세하게 조정하는데 사용한다.

○ setitimer()와 alarm()체계호출

Linux는 사용자방식프로쎄스가 간격시계(interval timer)라는 특별한 시계를 사용

할수 있게 한다. 이 시계는 주기적으로 프로세스에 Unix신호(3장 3절 참고)를 보내게 한다. 지정한 시간이 지나면 한번만 신호를 보내도록 간격시계를 설정할수도 있다. 따라서 각 간격시계는 다음과 같은 성질로 이루어진다.

· 신호를 보낼 빈도. 이 값이 빈값(null)이면 신호가 한번만 발생한다.

· 다음신호가 발생할 때까지 남은 시간

앞서 언급한 정확성문제는 이 시계에도 적용된다. 이 시계는 지정한 시간이 지난 후에 실행되는것은 확실하지만 언제 배달될것인지 정확하게 예측하는것은 불가능하다.

POSIX setitimer()체계호출을 사용하여 간격시계를 동작시킨다. 첫번째 파라메터는 다음방책중 어떤것을 채택할것인지 지정한다.

ITIMER_REAL

실제로 경과한 시간. 프로세스는 SIGALRM신호를 받는다.

ITIMER_VIRTUAL

프로세스가 사용자방식에서 보낸 시간. 프로세스는 SIGVTALRM신호를 받는다.

ITIMER_PROF

프로세스가 사용자방식과 핵심부방식모두에서 보낸 시간. 프로세스는 SIGPROF신호를 받는다.

각 방책에 따른 간격시계를 구현할 목적으로 프로세스서술자는 다음마당 세쌍을 포함한다.

· it_real_incr과 it_real_value

· it_virt_incr과 it_virt_value

· it_prof_incr과 it_prof_value

각쌍의 첫번째 마당은 두 신호사이의 간격을 틱크수로 보관하고 두번째 마당은 시계의 현재값을 저정한다. ITIMER_REAL간격시계는 프로세스가 CPU에서 실행중이지 않더라도 신호를 보내야 하므로 동적시계를 리용하여 구현한다. 따라서 각 프로세스서술자에는 real_timer라는 동적시계객체가 있다.

setitimer()체계호출은 real_timer마당을 초기화한 후 add_timer()를 호출해서 동적시계를 적절한 목록에 추가한다. 시계가 완료되면 핵심부는 it_real_fn()시계함수를 실행한다. 이 함수는 프로세스에 SIGALRM신호를 보낸다. it_real_incr이 0이 아니면 expires마당을 다시 설정한 후 시계를 다시 활성화한다.

ITIMER_VIRTUAL과 ITIMER_PROF간격시계는 프로세스가 실행중인 동안에만 갱신되므로 동적시계를 사용할 필요가 없다. PIT의 시계새치기조종기나(단일처리기) 국부시계새치기조종기에서(SMP) 호출하는 update_one_process()함수는 do_it_virt()와 do_it_prof()함수를 호출한다. 따라서 각 틱크가 발생할 때까지 두 간격시계를 갱신하며 시계가 완료되면 현재프로세스에 해당 신호를 보낸다.

alarm()체계호출을 지정한 시간간격이 지나면 이것을 호출한 프로세스에

SLGALRM신호를 보낸다. 이것은 ITIMER_REAL방책을 지정하여 setitimer()를 호출한것과 매우 비슷하며 마찬가지로 프로쎄스서술자에 있는 real_timer동적시계를 사용한다. 따라서 alarm()과 ITIMER_REAL을 지정한 setitimer()를 동시에 사용할수 없다.

제 3 절. 체계호출

조작체계는 사용자방식에 있는 프로쎄스가 CPU와 디스크, 인쇄기 등의 하드웨어장치와 호상작용할수 있도록 일련의 대면부를 제공한다. 이렇게 응용프로그람과 하드웨어사이에 별도의 계층을 두면 여러가지 우점이 있다.

먼저 프로그람작성이 쉬워지고 사용자가 하드웨어장치의 특성을 다루어야 하는 저수준프로그람을 배울 필요가 없다. 다음으로 핵심부는 사용자의 요구를 실제로 처리하기전에 대면부수준에서 요구한것이 옳바른지 검사할수 있어서 체계안전성이 높아진다.

끝으로 그러면서도 중요한 점으로 이러한 대면부는 프로그람을 똑같은 대면부집합을 제공하는 어떤 핵심부에서나 콤파일하고 제대로 실행할수 있게 하므로 프로그람호환성이 좋아진다. Unix체계는 핵심부에 요구를 하는 《체계호출(system call)》이라는 방법으로 사용자방식프로쎄스와 하드웨어장치사이 대부분의 대면부를 구현한다.

이 절에서는 Linux에서 사용자방식프로그람이 핵심부에 요구를 하는 체계호출을 어떻게 구현하는지 자세히 살펴보자.

1. POSIX API와 체계호출

먼저 응용프로그람프로그람대면부(API: application programming interface)와 체계호출의 차이를 살펴보자.

API는 주어진 봉사를 어떻게 받는지 지정하는 함수정의이고 체계호출은 쏘프트웨어새치기를 통해 핵심부에 하는 직접적인 요구이다. Unix체계에는 프로그람작성자에게 API를 제공하는 함수를 포함한 다양한 서고가 있다. libc표준C서고에서 정의하는 API중 일부는 체계호출을 진행하는 목적으로만 사용하는 《래퍼루틴(wrapper routine)》을 사용한다. 보통 각 체계호출마다 대응하는 래퍼루틴이 하나씩 있으며 래퍼루틴은 응용프로그람에서 사용할 API를 정의한다. 그러나 그 반대는 성립하지 않는다. API는 특정체계호출에 대응할 필요는 없다. 례를 들어 open체계호출을 실행할수 있도록 include/asm-i386/unistd.h파일에서 open()이라는 래퍼루틴을 제공하고 사용자방식프로그람은 이 open()을 API처럼 직접 호출하거나 fopen()같은 API를 사용하여 간접호출한다.

무엇보다도 API는 사용자방식내에서 봉사를 직접 제공할수 있다.(수학함수처럼 추상적인것은 체계호출을 사용할 리유가 없다.) 다음으로 API함수 하나는 여러 체계호출

을 사용할수 있다. 더우기 여러 API함수가 똑같은 체계호출을 사용하고 여기에 기능을 추가로 덧붙일수도 있다. 례를 들어 Linux의 malloc()과 calloc(), free() POSIX API를 libc서고에서 구현하는데 이 서고에 있는 코드에서 기억기할당과 해제를 직접 관리하며 brk()체계호출을 사용하여 프로쎄스동적기억의 크기를 늘이거나 줄인다.(3장의 《동적기억관리》참고)

POSIX 표준은 체계호출이 아닌 API를 규정한다. 응용프로그람에 정확한 API집합을 제공하기만 하면 해당 함수를 구현하는 방식에 상관없이 POSIX와 호환하는 체계라고 인정할수 있다. 실제로 Unix체계는 아니지만 사용자방식서고에서 전통Unix봉사를 제공하기때문에 POSIX호환인정을 받은 체계도 여러개 있다.

프로그람작성자관점에서 보면 API와 체계호출을 구별하는것은 무의미하다. 이것들에게는 단지 함수명과 파라메터 형, 결과값의 의미만이 중요할뿐이다. 그러나 핵심부설계자의 관점에서 보면 체계호출은 핵심부에 속하지만 사용자방식서고는 그렇지 않으므로 이러한 구별은 중요하다.

대부분의 래퍼루틴은 옹근수값을 되돌린다. 이 값의 의미는 해당 체계호출마다 다르다. 보통은 결과값이 1인 경우 핵심부가 프로쎄스의 요구를 처리할수 없음을 나타낸다.

체계호출조종기는 잘못된 파라메터나 자원부족, 하드웨어문제 등의 리유로 실패할수 있다. 구체적인 오유코드는 libc 서고에서 정의하는 errno변수에 들어있다. 각 오유코드를 정의 옹근수값을 마크로상수로 정의한다. POSIX표준에서는 몇가지 오유코드의 마크로이름을 규정한다. 80x86체계용Linux에서는 이 마크로를 include/asm-i386/errno.h 머리부파일에서 정의한다. Unix체계사이에서 C프로그람의 호환성을 위해 표준C서고머리부파일인 /usr/include/errno.h는 include/asm-i386/error.h 머리부파일을 포함한다.

다른 체계에서도 머리부파일을 모아둔 자신만의 특별한 보조등록부가 있다.

2. 체계호출조종기와 봉사루틴

사용자방식프로쎄스가 체계호출을 실행하면 CPU는 핵심부방식으로 전환해서 핵심부함수를 실행하기 시작한다. Linux에서는 int $0x80기호언어명령어를 실행하여 체계호출을 실행해야 한다. 이 명령어는 128번 벡토르를 가진 프로그람에 의한 례외를 발생시킨다.(5장 1절의 《새치기와 함정, 체계문》과 《새치기와 례외의 하드웨어적인 처리》참고)

핵심부는 서로 다른 많은 체계호출을 구현하므로 프로쎄스는 자신이 원하는 체계호출을 구별하는 《체계호출번호(system call number)》를 파라메터로 전달해야 한다.

이런 목적으로 eax등록기를 사용한다. 후에 《파라메터전달》에서 보지만 보통 체계호출을 진행할 때에는 추가로 파라메터를 전달한다.

모든 체계호출은 옹근수값을 반환한다. 체계호출의 결과값과 래퍼루틴의 결과값에

대한 관계는 서로 다르다. 핵심부에서는 정수로서 0은 체계호출이 성공적으로 마쳤음을 의미하고 부수는 오유를 의미한다. 오유가 발생한 경우 결과 값은 errno변수를 통해 응용프로그람에 반환해야 하는 오유코드를 부수로 만든것이다.

핵심부는 errno변수를 설정하거나 이 변수를 사용하지 않는다. 대신 래퍼루틴이 체계호출에서 돌아온 후 이 변수를 설정하는 일을 한다.

체계호출조종기는 다른 례외조종기와 구조가 비슷하며 다음과 같은 작업을 수행한다.

・대부분의 등록기내용을 핵심부방식탄창에 보관한다. (이 작업은 모든 체계호출에서 공통적이며 기호언어로 작성되여있다.)

・《체계호출봉사루틴(system call service routine)》이라는 C함수를 호출하여 체계호출을 처리한다.

・ret_from_sys_call()함수(역시 기호언어로 작성되여있다.)를 리용하여 조종기에서 빠져 나온다.

xyz라는 체계호출과 관련한 봉사루틴의 이름은 일부례외는 있지만 보통 sys_xyz()이다.

그림 8-6에 체계호출을 진행하는 응용프로그람과 해당하는 래퍼루틴, 체계호출조종기, 체계호출봉사루틴사이의 관계를 도식화하였다. 화살표는 함수사이의 실행흐름을 나타낸다.

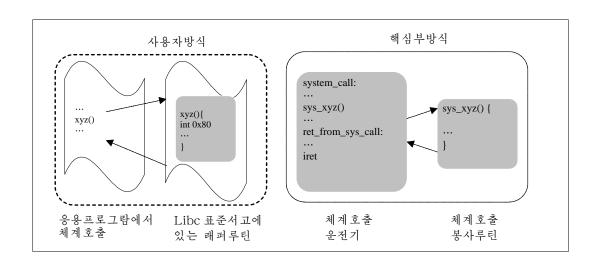



그림 8-6. 체계호출진행과정

핵심부는 체계호출번호와 해당 체계호출봉사루틴을 서로 련관시키기 위해 《체계호출분배표(system call dispatch table)》를 사용한다. 이 표는 sys_call_table배렬에 들어있고 입구점 NR_syscalls(보통 256)개를 포함한다.

n번째 입구점은 체계호출번호 n에 해당하는 체계호출봉사루틴주소를 보관한다.

NR_syscalls 마크로는 구현가능한 체계호출의 최대수를 지정하는 정적인 제한으로 실제 구현하고있는 체계호출수를 의미하지 않는다. 실제로 분배표에 있는 입구점중 어떤 것은 《구현하지 않은 (nonimplemented)》체계호출의 봉사루틴인 sys_ni_syscall() 함수의 주소를 보관한다. 이 함수는 단지 오유코드 -ENOSYS를 반환한다.

3. 체계호출초기화

핵심부를 초기화하는 과정에서 호출하는 trap_init()함수는 128번(즉 0x80)벡토르에 해당하는 《새치기서술자표(IDT: Interupt Desctiptor Table)》입구점을 다음과 같이 설정한다.

set_system_gate(0x80, &system_call);

이것은 문서술자(gate descriptor)의 마당을 다음값으로 지정한다.(5장의 《새치기와 함정, 체계문》참고)

토막선택기(Segment Selector)

핵심부코드토막의 토막선택기인 __KERNEL_CS

편위(Offset)

system_call()례외조종기에 대한 지적자

종류(Type)

15로 설정한다. 이것은 례외종류가 함정(trap)이고 해당 조종기는 마스크가능한 새치기를 금지하지 않음을 나타낸다.

서술자특권수준(DPL: Descriptor Privilege Level)

3으로 설정한다. 이것은 사용자방식에 있는 프로쎄스가 례외조종기를 호출할수 있게 한다.(5장의 《새치기와 례외의 하드웨어적인 처리》 참고)

4. system_call()함수

system_call()체계호출조종기를 구현하는 함수이다. 이 함수는 체계호출번호와 함께 례외조종기가 사용할수 있도록 조종장치가 자동으로 보관한(5장의 《새치기와 례외의 하드웨어적인 처리》 참고) eflags와 cs, eip, ss, esp등록기를 제외한 모든 CPU 등록기를 탄창에 보관한다. 5장에 있는 《새치기조종기를 위한 등록기보관》에서 설명한 SAVE_ALL마크로는 ds와 es를 각각 핵심부자료토막의 토막선택기로 설정한다.

```
system_call:
        pushl %eax
        SAVE_ALL
        movl $0xffffe000, %ebx /* or 0xfffff000 for 4-kB stacks */
        andl %esp, %ebx
```

이 함수는 ebx 등록기에 current 프로쎄스서술자를 보관한다. 이것은 핵심부탄창지적자를 8kB의 배수가 되도록 아래자리비트를 지워서 수행한다.(3장의 《프로쎄스구별하기》참고)

다음으로 system_call()함수는 current의 ptrace 마당에 PT_TRACESYS기발이 들어있는지 즉 오유수정기가 프로그람이 체계호출을 실행하는것을 추적하고있는지 검사한다. 이런 경우 system_call()은 syscall_trace()함수를 체계호출봉사루틴을 실행하기 직전에 한번, 직후에 한번, 총 두번을 호출한다. 이 함수는 current를 중단해서 오유수정을 하는 프로쎄스가 체계호출에 대한 정보를 수집할수 있게 한다.

다음으로 사용자방식프로쎄스가 전달한 체계호출번호가 옳바른지 검사한다. 이 번호가 NR_syscalls보다 크거나 같으면 체계호출조종기를 완료한다.

```
cmpl $(NR_syscalls), %eax
        jb nobadsys
        movl $(-ENOSYS), 24(%esp)
        jmp resume_userspace
nobadsys:
```

체계호출번호가 잘못된것이면 eax를 보관했던 탄창위치(현재탄창꼭대기부터 24번째 편위)에 ENOSYS값을 설정한다. 그리고 나서 resume_userspace()로 이행한다. 이렇게 해서 프로쎄스가 사용자방식에서 실행을 재개하면 eax에 보관한 부수결과값을 알게 된다.

마지막으로 eax에 들어있는 체계호출번호와 관련된 특정봉사루틴을 호출한다.

```
call *sys_call_table(0, %eax, 4)
```

분배표의 각 입구점은 4B이므로 핵심부는 체계호출번호에 4를 곱한 후 여기에 sys_call_table 분배표의 시작주소를 더한 후 표의 이 위치에 있는 봉사루틴에 대한 지적자를 얻어와서 호출할 봉사루틴의 주소를 알수 있다. 봉사루틴이 끝나면 system_call()은 eax에서 결과값을 얻어서 탄창에서 사용자방식의 eax등록기가 보관했던곳에 기록한다. 다음으로 체계호출조종기를 마치는 ret_from_sys_call()로 이행한다.(5장의 《ret_from_sys_call()함수》 참고)

```
movl  %eax, 24(%esp)
jmp ret_from_sys_call
```

사용자방식에서 실행을 재개한 프로쎄스는 eax등록기에서 체계호출의 결과값을 알수 있다.

5. 파라메터전달

보통 함수와 마찬가지로 체계호출도 종종 여러 입출력파라메터를 받는다. 이 파라메터는 실제값(수자)일수도 있고 사용자방식프로쎄스의 주소공간에 있는 변수, 심지어 사

용자방식내에 있는 함수를 가리키는 지적자를 담은 자료구조의 주소일수도 있다.

system_call()함수는 Linux에 있는 모든 체계호출의 공통적인 진입점이므로 체계호출은 적어도 eax등록기로 전달하는 체계호출번호라는 파라메터 하나를 받는다. 례를 들어 응용프로그람이 래퍼함수 fork()를 호출하면 int $0x80기호언어명령어를 실행하기 전에 eax등록기를 2(즉 __NR_fork)로 설정한다. 등록기는 libe서고에 들어있는 래퍼루틴에서 설정하므로 프로그람작성자는 보통 체계호출번호에 신경쓸 필요가 없다.

fork()체계호출은 다른 파라메터를 필요로 하지 않는다. 그러나 많은 체계호출이 파라메터를 추가로 받으며 응용프로그람에서 직접 전달해야 한다. 례를 들어 mmap()체계호출은(체계호출번호 외에도) 파라메터를 6개 받는다. 일반적인 C함수에서는 파라메터를 현재프로그람탄창에 기록하여(사용자방식탄창이든 핵심부방식탄창이든) 전달한다. 체계호출은 사용자방식에서 핵심부방식으로 넘어가는 특별한 함수이기때문에 사용자방식탄창은 물론 핵심부방식탄창도 사용할수 없다. 대신 int $0x80기호언어명령어를 실행하기 전에 체계호출파라메터를 CPU등록기에 보관한다. 체계호출봉사루틴은 일반C함수므로 핵심부는 이 함수를 호출하기 전에 CPU등록기에 보관한 파라메터를 핵심부방식탄창으로 복사한다.

왜 핵심부는 파라메터를 사용자방식탄창에서 핵심부방식탄창으로 바로 복사하지 않는가? 무엇보다도 동시에 두 탄창을 가지고 작업하는것은 복잡하다. 다음으로 등록기를 사용하면 체계호출조종기구조는 다른 례외조종기와 비슷해진다.

그러나 파라메터를 등록기로 전달하려면 다음 두 조건을 만족해야 한다.

・각 파라메터는 등록기크기인 32bit를 넘을수 없다.(이 내용은 64bit방식에는 해당하지 않는다.)

・Intel펜티움처리기의 등록기수는 매우 제한되여있으므로 파라메터의 개수는 6개를(eax등록기로 전달하는 체계호출번호를 포함하여) 넘을수 없다.

첫번째 조건은 항상 참이다. POSIX표준에 따르면 32bit등록기에 보관할수 없는 큰 파라메터는 반드시 지적자로 전달해야 하기때문이다. 대표적인 례로 64bit구조체 2개를 받는 settimeofday()체계호출이 있다. 그런데 두번째 조건과 관련하여 6개가 넘는 변수를 받는 체계호출도 있다. 이 경우 등록기 하나를 사용하여 파라메터값을 보관하는 프로세스주소공간내의 기억기령역을 가리키도록 한다.

물론 프로그람작성자는 이러한 일에 신경쓸 필요가 없다. 래퍼루틴을 호출하면 다른 C함수를 호출하는것과 마차가지로 파라메터를 자동적으로 탄창에 보관한다. 래퍼루틴은 핵심부에 파라메터를 전달하는 옳바른 방법을 알고있다. 체계호출파라메터를 보관하는 등록기 6개는 순서대로 eax(체계호출번호를 보관한다.), ebx, ecx, edx, esi, edi이다. 앞서 살펴본것처럼 system_call()은 SAVE_ALL마크로를 사용하여 이 등록기값을 핵심부방식탄창에 보관한다. 따라서 체계호출봉사루틴이 실행되여 탄창을 보면 먼저 system_call()의 되돌이주소가 있고 다음에 ebx(체계호출의 첫번째 파라메터), 그 뒤

로 ecx 등이 있음을 보게 된다.(5장의 《새치기조종기를 위한 등록기보관》참고) 이 탄창구성은 일반적인 함수호출과 완전히 일치한다. 따라서 봉사루틴은 일반C언어구조를 사용하여 파라메터를 참조할수 있다.

실례를 하나 들어보자. write()체계호출을 처리하는 sys_write()봉사루틴은 다음과 같이 선언한다.

ssize_t sys_write(unsigned int fd, const char __user *buf, size_t count)

C콤파일러는 탄창의 맨우부터 시작해서 돌아갈 주소 바로 다음에 fd와 buf, count파라메터가 있을것이라고 생각하고 기호언어함수를 만든다. 이 파라메터의 위치는 각각 ebx, ecx, edx 등록기를 보관한 곳이다. 드문 경우지만 체계호출이 어떠한 파라메터도 사용하지 않는 경우에도 해당 봉사루틴이 체계호출이 호출되기 직전에 CPU등록기의 내용을 알아야 하는 경우가 있다. 이런 실례로 fork()를 구현하는 do_fork()함수가 있는데 체계호출을 실행할 당시의 등록기값을 알아야 자식프로쎄스의 thread마당으로 이 등록기를 복제할수 있다.(3장의 《threade마당》참고) 이 경우 봉사루틴은 pt_regs형파라메터를 통해 SAVE_ALL마크로가 핵심부방식탄창에 보관한 값에 호출할수 있다.(5장의 《do_IRQ()함수》 참고)

int sys_fork(struct pt_regs regs)

봉사루틴의 결과값은 eax등록기에 기록해야 한다. 이 작업은 C콤파일러가 return: 명령을 실행할 때 자동으로 수행한다.

6. 파라메터확인

핵심부는 사용자의 요구를 처리하기 전에 모든 체계호출파라메터를 주의깊게 검사해야 한다. 어떤 종류의 검사를 할지는 체계호출과 특정파라메터에 따라 다르다. 앞서 소개한 write()체계호출을 생각해보자. fd파라메터는 특정파일을 서술하는 파일서술자이여야 한다. 따라서 sys_writ()는 fd가 이전에 열어놓은 파일의 서술자가 맞는지, 프로쎄스가 해당 파일에 쓰기를 할수 있는지 검사해야 한다. 이 조건중 하나라도 맞지 않으면 조종기는 부수값을, 이 경우에는 오유코드 EBADF를 반환해야 한다. 그런데 모든 체계호출에서 공통적인 검사류형이 하나 있다. 파라메터가 주소인 경우 핵심부는 이 주소가 프로쎄스주소공간내에 있는지 검사해야 한다. 이런 검사를 하는 방법은 두가지이다.

·선형주소가 프로쎄스주소공간에 속하는지, 속하면 이 주소를 포함하는 기억기령역이 해당 호출권한이 있는지 확인한다.

·선형주소가 PAGE_OFFSE보다 낮은지(즉 핵심부용으로 예약한 주소범위에 들어가지 않는지)만 확인한다.

초기 Linux핵심부는 첫번째 류형의 검사를 하였다. 그러나 체계호출에 전달한 각 주소파라메터를 모두 검사해야 하므로 퍼그나 시간이 걸린다. 더구나 이런 잘못을 하는 프로그람은 매우 드물기때문이 보통 이러한 검사는 무의미하다. 따라서 2.2핵심부부터

Linux는 두번째 방법으로 검사를 하는데 프로쎄스기억기령역서술자를 검색할 필요가 없기때문에 훨씬 더 효률적이다.

선형주소가 PAGE_OFFSET보다 작다는것은 해당 주소가 옳다는 필요조건이지만 충분조건은 아니다. 그러나 다른 오유를 나중에 발견할것이므로 핵심부에서 이 정도의 검사만 한다고 위험하지는 않다.

따라서 이 호출방법은 실질적인 검사를 가능한 마지막 순간까지 즉 페지화장치가 선형주소를 물리주소로 바꾸는 순간까지 미루는것이다. 뒤에 나오는 《동적주소검사: 수선코드》에서 페지절환례외조종기가 사용자방식에서 파라메터로 전달하여 핵심부방식에서 사용한 잘못된 주소를 검출하는 방법을 설명한다. 어떤 사람은 도대체 왜 이런 초보적인 검사가 필요한지 의아해 할것이다. 이런 종류의 검사는 실제로 프로쎄스주소공간과 핵심부주소공간을 잘못된 호출로부터 보호하는데 필수적이다.

4장에서 본것처럼 주기억기는 PAGE_OFFSET부터 시작하여 사영(mapping)된다. 이것은 핵심부코드가 기억기에 존재하는 모든 페지에 호출할수 있음을 의미한다. 따라서 이런 초보적인 검사라도 하지 않으면 사용자방식프로쎄스가 핵심부주소공간에 속하는 주소를 파라메터로 전달하여 페지절환례외를 일으키지 않고 기억기에 존재하는 페지에 값을 읽거나 쓸수 있게 된다.

체계호출에 전달한 주소는 access_ok()마크로를 통하여 수행된다. 이 함수는 addr과 size라는 두 파라메터를 통해 작업한다. 이 함수는 addr에서 addr+size-1사이에 있는 주소구간을 검사한다.

이 함수는 먼저 가장 높은 주소인 addr+size가 $2^{32}-1$보다 큰지 검사한다.

GNU C콤파일러(gcc)는 unsigned long옹근수자료형과 지적자를 모두 32bit 수자로 나타내므로 이것은 자리넘침조건을 검사하는것과 같다. 이 함수는 또한 addr+size가 current의 addr+limit.seg마당에 보관된 값보다 큰지 검사한다.

이 마당은 보통 일반프로쎄스에서는 PAGE_OFFSET, 핵심부스레드에서는 0xffffffff이다. addr_limit.seg마당은 get_fs와 set_fs마크로를 리용하여 동적으로 바꿀수 있다.

이 마크로는 핵심부가 체계호출봉사루틴을 직접 호출하거나 핵심부자료토막에 있는 주소를 파라메터로 넘길수 있게 한다. 이 마크로는 다음과 같다.

```
int access_ok(const void * addr, unsigned long size)
{
    unsigned long a = (unsigned long) addr;
    if (a + size < a ||
        a + size > current_thread_info( )->addr_limit.seg)
        return 0;
    return 1;
```

}

verify_area()함수는 access_ok()마크로와 같은 기능을 수행하는데 원천코드에서는 이 함수를 많이 리용한다.

핵심부코드에서는 사용자방식에 있는 프로쎄스의 주소공간에 직접 호출하면 안된다. 그래서 다음에 나오는 프로쎄스주소공간을 호출하기 위한 여러 함수와 마크로를 제공한다. 그리고 체계호출봉사루틴이나 장치구동프로그람의 여러 연산함수를 포함한 많은 함수는 자신에 전달된 지적자가 사용자방식의 주소공간일것이라고 생각하고 주소검사를 한다. 핵심부에서 이런 함수를 직접호출하면 verify_area() 등의 함수를 사용하여 주소를 검사할 때 주소가 PAGE_OFFSET(즉 current->addr_limit.seg)보다 크기때문에 잘못된 주소로 여기게 된다. 이것을 피하려면 주소공간의 범위를 핵심부주소공간의 범위로 설정해야 한다. 이것이 get_fs()와 set_fs()를 사용하는 리유이다. 이 마크로는 각각 current->addr_limit을 읽고 쓴다. 핵심부코드에서 체계호출조종기를 호출하기 전에 핵심부는 mm_segment_t oldfs=get_fs()로 현재범위를 가져와 보관하고 set_fs(KERNEL_DS)를 통해 범위를 핵심부의 범위로 설정한다. 그리고 체계호출을 실행한 후에는 set_fs(oldfs)를 호출해서 이전범위로 복구한다.

핵심부 2.6부터는 주소를 검사하는 함수가 다음과 같이 되여있다.

```
        static inline int verify_area(int type, const void __user * ad
      dr, unsigned long size)
{
        return access_ok(type,addr,size) ? 0 : -EFAULT;
}
```

type은 %VERIFY_READ이나 %VERIFY_WRITE과 같은 호출의 형을 나타낸다. addr는 검사해야 하는 리용자공간의 시작블로크주소이며 size는 검사하는 블로크의 크기를 나타낸다. 리용자주소공간에 있는 기억블로크에 대한 지적자가 유효한가를 검사하는데 유효하면 0을 되돌리며 무효하면 EFAULT를 되돌린다. 이 함수는 access_ok()에 의해 교체된다.

초기의 Linux핵심부에서는 핵심부방식에서 사용자주소공간에 호출하기 위해 핵심부방식에 진입하면 사용자프로쎄스의 자료토막(ds)를 fs등록기로 보관하였다. 따라서 핵심부코드에서는 fs등록기를 통해서 사용자주소공간에 호출할수 있었다. get_fs()와 set_fs()의 명칭과 기능은 여기서 유래하며 지금은 fs등록기를 사용하지 않지만 핵심부코드내에서 체계호출조종기를 호출할 때에는 같은 일을 해야 한다.

access_ok마크로는 verify_area()와 똑같은 검사를 한다. 둘의 차이는 결과값이다. access_ok마크로는 지정한 주소구간이 유효이면 1, 그렇지 않으면 0을 반환한다. __addr_ok마크로 역시 지정한 선형주소가 옳바르면 1, 아니면 0을 반환한다.

7. 프로쎄스주소공간호출

체계봉사루틴이 프로쎄스의 주소공간에 들어있는 자료를 읽거나 써야 하는 경우가 많다. Linux는 이런 호출을 쉽게 하려고 일련의 마크로를 제공한다. 이중에서 get_user()와 put_user()를 설명하겠다. 전자는 지정한 주소부터 1이나 2, 4B의 련속된 자료를 읽을 때 사용하며 후자는 그 주소로 1이나 2, 4B의 자료를 써넣을 때 사용한다.

각 함수는 전송할 값인 x와 변수 ptr이라는 두 파라메터를 받는다. ptr은 몇B를 전송할지도 결정한다. get_user(x, ptr)에서 ptr가 가리키는 변수의 크기에 따라 이 함수는 __get_user_1()이나 __get_user_2(), __get_user_4()기호언어함수로 확장된다. 이중 하나인 __get_user_2()를 례로 들어보자.

```
__get_user_2:
        addl $1, %eax
        jc bad_get_user
        movl %esp, %edx
        andl $0xffffe000, %edx
        cmpl 12(%edx), %eax
        jae bad_get_user
2: movzwl -1(%eax), %edx
        xorl %eax, %eax
        ret
bad_get_user:
        xorl %edx, %edx
        movl $-EFAULT, %eax
        ret
```

eax등록기는 읽어들일 첫번째 바이트의 주소 ptr을 보관한다. 처음 6개의 명령어는 실질적으로 verify_area()함수와 똑같은 검사를 한다. 이것은 읽어 들일 2B의 주소가 4GB와 current프로쎄스의 addr_limit.seg마당(이 마당은 프로쎄스서술자의 편위 12에 들어있고 cmpl 명령어의 첫번째 피연산자이다.)보다 작은지 확인한다.

주소가 유효하면 이 함수는 movzwl명령어를 실행하여 읽은 자료를 edx등록기의 아래자리 2B에 보관하며 edx의 웃자리바이트를 0으로 설정한다. 그리고 나서 eax에 결과값 0을 보관하고 완료한다. 주소가 유효하지 않으면 edx를 0으로 만들고 EFAULT값을 eax에 보관한 후 완료한다.

put_user(x,ptr)마크로는 x값을 ptr부터 시작하는 프로쎄스주소공간에 넣는다는 점을 제외하면 앞에서 설명한것과 비슷하다. x의 크기에 따라 __put_user_asm()마크로

(1이나 2, 4B)나 __put_user_u64()마크로(8B)를 호출한다. 두 마크로 모두 성공한 경우 eax에 0을 넣어서 반환하며 실패한 경우 EFAULT를 반환한다.

표 8-12에 핵심부방식에서 프로쎄스주소공간에 호출할수 있는 여러 함수와 마크로를 렬거하였다 이중 몇개에는 밑선 두개(__)가 앞붙이로 붙은 변형이 있다. 앞붙이가 붙지 않은 함수는 요구한 선형주소구간이 유효한지 검사하는데 시간을 소요하지만 앞붙이가 붙은 함수는 이런 검사를 하지 않는다. 핵심부가 프로쎄스주소공간에 똑같은 기억기령역에 반복해서 호출해야 하는 경우에는 시작할 때 한번만 검사하고 나중에는 검사하지 않는것이 훨씬 효률적일것이다.

표 8-12. 프로쎄스의 주소공간에 호출하는 함수와 마크로

함 수	동작
get_user __get_user	사용자공간에서 옹근수값을 읽어들인다.(1이나 2, 4B)
put_user __put_user	사용자공간으로 옹근수값을 쓴다.(1이나 2, 4B)
copy_from_user __copy_from_user	사용자공간에서 원하는 크기만큼 차단을 복사한다.
copy_to_user __copy_to_user	사용자공간으로 원하는 크기만큼 차단을 복사한다.
strncpy_from_user __strncpy_from_user	사용자공간에 있는 0으로 끝나는 문자렬을 복사한다.
strlen_user strnlen_user	사용자공간에 있는 0으로 끝나는 문자렬의 길이를 반환한다.
clear_user __clear_user	사용자공간에 있는 기억기령역을 0으로 채운다.

동적주소검사: 수선코드

앞서 살펴본것처럼 verify_area()함수와 access_ok, __addr_ok마크로는 체계호출에 파라메터로 전달한 선형주소에 대해 초보적인 검사만 한다. 이것으로는 해당 주소가 프로쎄스주소공간에 들어있다고 담보할수 없으므로 잘못된 주소를 넘겨준 프로쎄스는 페지절환례외를 발생시킬수 있다.

어떻게 핵심부가 이런 종류의 오유를 발견하는지 설명하기 전에 핵심부방식에서 페

지절환례외가 일어날수 있는 4가지 경우를 살펴보자.

1. 핵심부가 프로쎄스주소공간에 속한 주소에 호출할 때 해당 페지틀이 존재하지 않거나 읽기전용페지에 쓰려고 한 경우이다. 이 경우 조종기는 새 페지틀을 할당하여 이것을 초기화해야 한다.(3장의 《요구페지화》와 《쓰기복사》참고)

2. 핵심부가 프로쎄스주소공간에 속한 페지를 호출할 때 해당 페지표입구점이 초기화되지 않은 경우이다.(3장의 《불련속적인 기억기령역호출》참고) 이 경우 핵심부는 현재프로쎄스의 페지표에 있는 일부입구점을 제대로 설정해야 한다.

3. 어떤 핵심부함수에는 해당 프로그람을 실행하면 례외를 일으킬수 있는 프로그람오유가 있을수 있다. 아니면 일시적인 하드웨어오유로 례외가 발생할수 있다. 이 경우 조종기는 핵심부웁스(kernel oops)를 실행해야 한다.(3장의 《주소공간밖의 잘못된 주소처리》 참고)

4. 이 장에서 소개한 경우로 체계호출봉사루틴이 체계호출에 전달한 주소의 기억기령역에 읽거나 쓰기를 할 때 해당 주소가 프로쎄스주소공간에 들어있지 않은 경우이다.

페지절환조종기는 잘못된 선형주소가 프로쎄스소유의 기억기령역중 하나에 들어있는지 검사하여 첫번째 경우는 쉽게 인식할수 있다. 두번째 경우 역시 프로쎄스의 페지표에 해당 주소를 배치하는 빈값이 아닌 옳바른 입구점이 있는지 검사해서 알수 있다. 그러면 조종기가 남은 두 경우를 어떻게 구별하는지 살펴보자.

8. 례외표

페지절환의 원인을 알아내는 열쇠는 핵심부가 프로쎄스주소공간에 호출하는데 사용하는 함수의 작은 부분에서 오유가 발생한다는데 있다. 프로쎄스주소공간에 호출하는데는 앞에서 설명한 적은 수의 함수와 마크로만을 사용한다. 따라서 례외가 잘못된 파라메터때문에 발생하였다면 이것을 발생시킨 명령어는 이 함수중 하나에 들어있거나 이 마크로중 하나가 확장되여만들어진 코드안에 있을것이다. 사용자공간을 호출하는 명령어의 개수는 매우 적다.

그러므로 프로쎄스주소공간에 호출하는 각 핵심부명령의 주소를 《례외표(exception table)》라는 구조체로 모으는 일은 그리 어렵지 않다.

이 작업만 제대로 하면 나머지는 쉽다. 핵심부방식에서 페지절환례외가 발생하면 do_page_fault()조종기는 례외표를 검사한다. 례외를 일으킨 명령어의 주소가 이곳에 들어있으면 잘못된 체계호출파라메터때문에 오유가 발생한것이고 그렇지 않으면 더 심각한 다른 오유때문이다.

Linux는 여러 례외표를 정의한다. 핵심례외표는 핵심부프로그람영상을 만들 때 C 콤파일러가 자동으로 생성한다. 이것은 핵심부코드토막의 __ex_table단락에 보관되고

시작과 끝위치는 C콤파일러가 만드는 두 기호 __start___ex_table과 __stop___ex_table로 알수 있다.

나가서 동적으로 적재한 각 핵심부모듈은 자신만의 국부레외표를 포함한다. 이 표는 모듈영상을 만들 때 C콤파일러가 자동으로 생성하며 모듈을 실행중인 핵심부에 추가할 때 기억기에 적재된다.

레외표의 각 입구점은 다음 두 마당을 포함하는 exception_table_entry구조체이다.

insn

프로쎄스주소공간에 호출하는 명령어의 선형주소

fixup

insn에 있는 명령어에서 페지절환레외가 발생할 때 호출할 기호언어코드의 주소

《수선코드(fixup code)》는 레외때문에 발생한 문제를 해결하는 기호언어명령어 몇개로 이루어진다. 이 절뒤부분에서 나오지만 일반적으로 이런 수선작업은 강제로 봉사루틴이 사용자방식프로쎄스에 오유코드를 반환하도록 하는 일련의 명령어를 삽입하는것이다.

삽입할 명령어는 보통 프로쎄스주소공간에 호출하는 마크로나 함수내에서 정의한다. 때로는 콤파일러가 핵심부코드토막중 .fixup이라는 별도의 단락에 두기도 한다.

모든 레외표에 지정한 주소가 있는지 찾을 때에는 search_exception_table()함수를 사용한다.

이 함수는 주소가 표에 들어있으면 해당하는 fixup주소를, 없으면 0을 반환한다.

따라서 페지절환조종기 do_page_fault()는 다음과 같은 문장을 실행한다.

```
if ((fixup = search_exception_table(regs->eip)) != 0) {
        regs->eip = fixup;
        return;
}
```

regs->eip마당은 레외가 발생할 때 핵심부방식탄창에 보관한 eip등록기의 값이다. 이 등록기에 있는 값이(명령어지적자(instruction pointer)) 레외표에 들어있으면 do_page_fault()는 보관된 eip값을 search_exception_table()이 반환하는 주소로 바꾼다. 그리고 나서 페지절환조종기가 완료하면 새치기된 프로그람은 수선코드에서 실행을 재개하게 된다.

9. 레외표생성과 수선코드

프로그람작성자는 GNU기호언어의 .section지시어(directive)를 사용하여 다음에 나오는 코드를 실행파일의 어느 단락 (section)에 넣을지 지정할수 있다. 4절에서 보지만 실행파일에는 코드토막이 있고 이것은 다시 여러 단락으로 쪼개진다. 따라서 다음기

호언어명령어는 례외표에 입구점을 추가한다. 《a》속성은 이 단락을 나머지핵심부영상과 함께 기억기에 적재해야 한다고 지시한다.

```
.section __ex_table, "a "
        .long 잘못된 명령어주소, 수선코드주소
.previous
```

.previous지시어는 기호언어가 다음에 나오는 코드를 마지막 section지시어가 나오기 전 단락에 넣도록 지시한다. 다시 앞에서 언급한 __get_user_1()과 __get_user_2(), __get_user_4() 함수를 살펴보자. 프로쎄스주소공간에 호출하는 명령어에는 각각 1, 2, 3 이라는 표식이 붙어있다.

```
   __get_user_1:
        [...]
1:         movzbl (%eax), %edx
        [...]
__get_user_2:
        [...]
2:         movzwl -1(%eax), %edx
        [...]
__get_user_4:
        [...]
3:         movl -3(%eax), %edx
        [...]
bad_get_user :
        xorl %edx, %edx
        movl $-EFAULT, %eax
        ret

.section __ex_table, "a"
        .long 1b, bad_get_user
        .long 2b, bad_get_user
        .long 3b, bad_get_user
.previous
```

각 례외표입구점은 두 표식로 구성된다. 첫번째는 《앞》에 나오는(다른 말로 프로그람의 앞행에 나오는)표식임을 의미하는 b라는 접미사가 옹근수표식에 붙은것이다.

두번째는 수선코드의 주소이다. 수선코드는 이 세 함수에서 공통으로 사용하며 bad_get_user라는 표식이 붙어있다. 1이나 2, 3 표식이 붙은 명령어에서 페지절환례외

가 발생하면 이 수선코드를 실행하며 이 코드는 체계호출을 진행한 프로세스에 EFAULT오유코드를 되돌린다.

사용자방식주소공간에서 동작하고 수선코드기법을 사용하는 다른 핵심부코드의 실례로 strlen_user(string)마크로를 살펴보자. 이 마크로는 프로쎄스주소공간에 있는 0으로 끝나는 문자렬의 길이를 반환하며 오유가 발생한 경우 0을 반환한다. 이 마크로는 다음과 같은 기호언어명령어를 생성한다.

```
movl $0, %eax
        movl $0x7fffffff, %ecx
        movl %ecx, %ebp
        movl string, %edi
0:         repne: scasb
        subl %ecx, %ebp
        movl %ebp, %eax
1:
. section .fixup, “a”
2:         movl $0, %eax
        jmp 1b
.previous
.section __ex _table,” a”
        .long 0b, 2b
.previous
```

ecx와 ebp 등록기를 사용자방식주소공간에서 문자렬의 최대허용길이를 나타내는 0x7fffffff 값으로 초기화한다. repne; scasb 기호언어명령어는 반복해서 edi등록기가 가리키는 문자렬을 검색하여 eax에 들어있는 값 0(문자렬의 끝을 나타내는 \0문자)을 찾는다. scasb명령어를 반복할 때마다 ecx등록기의 값이 줄어들기때문에 결국 eax등록기에는 문자렬에서 검색한 바이트수, 즉 문자렬길이가 들어간다.

이 마크로의 수선코드는 .fixup단락에 들어간다. 《ax》속성은 이 단락을 기억기에 적재해야 하며 이 단락에 실행가능한 코드가 들어있음을 지시한다. 표식 0에 있는 명령어에서 페지절환례외가 발생하면 표식 2에 있는 수선코드를 실행한다. 이 코드는 단지 eax 등록기에 0을 보관하여 마크로가 문자렬의 길이 대신 오유코드 0을 반환하도록 한 후 마크로뒤에 나오는 명령어를 가리키는 표식 1로 이행한다.

10. 핵심부래퍼루틴

체계호출은 주로 사용자방식프로쎄스에서 사용하지만 서고함수를 사용할수 없는 핵심부스레드에서도 사용할수 있다. Linux는 체계호출에 해당하는 래퍼루틴(wrapper

routine)을 쉽게 선언할수 있게 _syscall0에서 _syscall6까지 마크로 7개를 정의한다.

마크로이름에 있는 수자 0-6은 체계호출이 사용하는 파라메터의 개수(체계호출번호 제외)를 말한다. 이 마크로를 libc표준서고에 들어있지 않은 래퍼루틴을 선언하는데도 사용할수 있다.(례를 들면 서고에서 Linux체계호출을 지원하지 않아서) 그러나 6개가 넘는 파라메터(체계호출번호 제외)를 받는 체계호출이나 비표준적인 결과값을 반환하는 체계호출용 래퍼루틴을 정의하는데는 사용할수 없다.

각 마크로는 파라메터를 정확히 2+2×n(n은 체계호출로 전달하는 파라메터의 개수)개 받는다. 처음 두 파라메터는 반환할 형과 체계호출의 이름이다. 추가로 전달하는 파라메터의 쌍은 해당 체계호출파라메터의 형과 이름이다. 례를 들어 fork()체계호출의 래퍼루틴은 다음과 같이 만들수 있다.

```
_syscall0(int, fork)
```

반면 write()체계호출의 래퍼루틴은 다음과 같이 만들수 있다.

```
_syscall3(int, write, int, fd, const char *, buf , unsigned int, count)
```

이 마크로는 다음과 같은 코드를 만들어낸다.

```
int writ (int fd, const char *buf, unsigned int count)
{
        long _ res:
        asm( "int $0x80"
                    :       "   =   a   "   (   __   r   e   s   )
                  : " 0"  (_NR _ write),  "b" ((long)fd),
                "c" ((long)buf),  "d" ((long)count));
        if ((unsigned long) __res >= (unsigned long)-125)  {
                errno = -__res;
                __res = -1;
        }
        return (int) __res:
}
```

__NR_writ 마크로는 _syscall3의 두번째 파라메터에서 만든것이다. 이것은 write()의 체계호출번호로 확장된다. 우의 함수를 콤파일하면 다음기호언어코드가 만들어진다.

```
write:
        pushl  %ebx                   ; ebx를 탄창에 보관한다.
        movl 8(%esp), %ebx            ; 첫번째 파라메터를 ebx에 보관한다.
        movl 12(%esp), %ecx  ; 두번째 파라메터를 ecx에 보관한다.
        movl 16(%esp), %edx           ; 세번째 파라메터를 edx 에 보관한다.
```

```
        movl $4, %eax          ; __NR_write를 eax에 보관한다.
        int $0x80              ; 체계호출을 실행한다.
        cmpl $-126, %eax       ; 결과값을 검사한다.
        jbe .L1                ; 오유가 발생하지 않았으면 이행한다.
        negl %eax              ; eax값의 부호를 바꾼다.
        movl %eax, errno       ; 결과값을 errno에 보관한다.
        movl $-1, %eax         ; eax를  1로 설정한다.
.L1:    popl %ebx              ; 탄창에서 ebx를 복구한다.
        ret                    ; 자신을 호출한 프로그람으로 돌아간다.
```

int $0x80명령어를 실행하기 전에 write()함수의 파라메터를 CPU등록기로 보관하는 방법을 눈여겨보자. eax로 반환하는 값이 1에서 125사이에 있는 경우(핵심부는 include/asm-i386/errno.h에서 정의하는 가장 큰 오유코드가 125라고 가정한다.) 오유코드로 해석해야 한다. 이 경우 래퍼루틴은 eax 값을 errno에 보관하고 1을 반환한다. 그렇지 않으면 eax의 값을 반환한다.

제4절. 프로그람실행

3장에서 설명한 《프로쎄스》의 개념은 초창기 Unix시절부터 실행중인 프로그람그룹이 체계자원을 두고 경쟁하는 행동을 나타내는데 사용해왔다. 여기서는 프로그람과 프로쎄스의 관계에 중점을 둔다. 특히 프로그람파일의 내용에 따라 핵심부가 프로쎄스의 실행문맥(execution context)을 설정하는 방법을 설명한다. 명령어들을 기억기에 읽어들여 CPU에 가리키는것이 그렇게 큰 문제로 보이지는 않지만 핵심부는 일부 부분에서 유연성을 제공해야 한다.

여러가지 형식의 실행파일

Linux는 다른 조작체계용으로 콤파일된 2진파일을 실행할수 있는 능력이 뛰여나다.

공유서고

많은 실행파일이 프로그람을 실행하는데 필요한 모든 코드를 포함하지 않고 핵심부가 실행시간에 서고에서 함수를 읽어들이기를 바란다.

실행문맥의 다른 정보

여기에는 프로그람작성자에게 익숙한 명령행파라메터와 환경변수를 포함한다. 프로그람은 디스크에 실행파일(executable file)로 보관된다. 실행파일은 실행할 함수의 목적코드와 함수가 사용할 자료를 포함한다. 프로그람의 많은 함수는 모든 프로그람작성자가 사용할수 있는 봉사루틴이다. 이것들의 목적코드는 《서고》라는 특별한 파일에 포함되여있다. 실제로 서고함수코드는 실행파일에 정적으로 복사할수도 있고(정적서고), 실행중에 프로쎄스에 련결할수도 있다.(공유서고, 여러 독립적인 프로쎄스가 공유서고의 코드를 공유할수 있다.)

프로그람을 시작할 때 사용자는 프로그람의 실행방식에 영향을 줄 두가지 정보를 제공한다. 명령행인자(Command-line argument)는 사용자가 쉘재촉문에서 실행파일명뒤에 직접 입력한다. HOME이나 PATH와 같은 환경변수(Environment variable)는 쉘에서 물려받지만 사용자가 프로그람시작전에 이 변수의 값을 변경할수 있다.

《실행파일》부분에서는 프로그람의 실행문맥를 설명한다. 《실행가능한 형식》부분에서는 Linux가 지원하는 실행가능한 몇가지 형식을 살펴본다. 또한 Linux가 다른 조작체계에서 콤파일한 프로그람을 실행하기 위해 《특성(personality)》을 어떻게 바꿀수 있는지 살펴본다. 마지막으로 《exec계렬함수》부분에서는 프로쎄스가 새로운 프로그람을 시작할수 있게 해주는 체계호출을 설명한다.

1. 실행파일

이미 프로쎄스를 실행문맥(execution context)으로 정의하였다. 즉 특정계산을 실행하는데 필요한 정보를 모아놓은것을 의미한다. 여기에는 호출하는 페지, 열린파일, 하드웨어등록기의 내용 등이 포함된다. 실행파일은 새로운 실행문맥를 어떻게 초기화하는지(즉 새로운 계산을 어떻게 시작하는지) 나타내는 일반파일이다.

사용자가 현재등록부의 파일목록을 보려고 한다고 가정하자. 사용자는 쉘재촉문에 외부명령 /bin/ls의 파일명을 입력하여 얻을수 있다는 사실을 알고있다.

명령쉘은 새로운 프로쎄스를 생성(fork)하고 새로운 프로쎄스는 execve()체계호출을 실행하면서(《exec 계렬함수》참고) ls실행파일의 전체 경로명을 포함하는 문자렬(이 경우는 /bin/ls)을 파라메터로 전달한다. sys_execve()봉사루틴은 대응하는 파일을 찾고 실행파일의 형식을 검사한 다음 그 안에 보관된 정보에 따라 현재프로쎄스의 실행문맥를 변경한다. 결과적으로 체계호출이 완료하면 프로쎄스는 실행파일에 보관된 코드를 실행하여 등록부목록을 렬거하는 작업을 수행한다.

프로쎄스가 새로운 프로그람의 실행을 시작하면 이전계산과정에서 얻은 자원을 버리기때문에 프로쎄스의 실행문맥이 크게 바뀐다. 앞의 실례에서 프로쎄스가 /bin/ls를 실행하기 시작하면 프로쎄스는 쉘인자를 execve()체계호출에 전달된 파라메터로 교체하고 새로운 쉘환경변수(《명령행인자와 쉘환경》 참고)를 얻는다. 부모프로쎄스로부터 물려받은(그리고 쓰기복사(Copy On Writ)기구를 통해 공유된) 모든 페지를 해제하고 새로운 사용자방식주소공간을 리용해서 새로운 계산을 시작한다. 프로쎄스의 권한까지도 바뀔수 있다.(아래 《프로쎄스의 자격과 특질》참고) 그러나 프로쎄스 PID는 바뀌지 않으며 새로운 계산은 이전계산에서 execve()함수를 실행하는 동안 자동으로 닫히지 않은 열린 파일의 서술자를 물려받는다.

2. 프로쎄스의 믿음권한

전통적으로 Unix체계에서는 각 프로쎄스에 몇가지 《믿음권한 (credential)》을

대응시킬수 있다. 믿음권한은 프로쎄스를 특정사용자와 특정사용자그룹에 련결(bind)한다. 믿음권한은 다중사용자체계에서 중요한데 믿음권한으로 각 프로쎄스가 할수 있는 일과 할수 없는 일을 결정하므로 각 개인자료의 완전성(integrity)과 전체적인 체계의 안정성(stability)을 보장해주기때문이다.

믿음권한을 사용하려면 프로쎄스자료구조와 보호할 자원 모두의 지원이 필요하다. 명백한 자원은 파일이다. Ext2파일체계에서 각 파일은 특정사용자가 소유하고 일부사용자의 그룹에 속한다. 파일의 소유자는 그 자신, 파일사용자그룹, 다른 모든 사용자로 구분하여 해당 파일에 어떤 종류의 연산을 허용할지 결정할수 있다. 프로쎄스가 파일에 호출하려고 시도하면 VFS는 파일의 소유자와 프로쎄스의 믿음권한으로 성립된 권한에 따라 해당 호출을 허용할지 검사한다.

프로쎄스의 믿음권한은 표 8-13에 나타낸 프로쎄스서술자의 여러 마당에 보관한다.

이 마당들은 체계의 사용자와 사용자그룹의 식별자를 포함하며 호출하려는 파일의 색인마디안에 보관된 대응하는 식별자와 비교된다.

표 8-13. 전통적인 프로쎄스믿음권한

이름	설명
uid, gid	사용자와 그룹의 실제(real)식별자
euid, egid	사용자와 그룹의 유효(effective)식별자
fsuid, fsgid	사용자와 그룹의 파일호출을 위한 유효식별자
groups	부가적이 그룹식별자
suid, sgid	사용자와 그룹의 보관된(saved) 식별자

UID 0은 초사용자(뿌리)를 나타내고 GID 0은 뿌리그룹을 나타낸다. 프로쎄스의 믿음권한이 0이면 핵심부는 권한검사를 하지 않고 이 특권프로쎄스에 체계관리나 하드웨어처리와 같이 일반 프로쎄스에 허가하지 않는 모든 일을 허가한다.

프로쎄스를 생성하면 해당 프로쎄스는 항상 부모의 믿음권한을 물려받는다. 그러나 프로쎄스가 새로운 프로그람의 실행을 시작하거나 적절한 체계호출을 통해 이 믿음권한을 나중에 변경할수 있다. 일반적으로 한 프로쎄스의 uid, euid, fsuid, suid마당값은 같다. 그러나 프로쎄스가 setuid프로그람 즉 setuid기발을 1로 설정한 실행파일을 실행하면 euid와 fsuid마당을 파일소유자의 식별자로 설정한다. 대부분의 검사는 이 두 마당을 대상으로 한다. fsuid는 파일관련연산에 사용하고 euid는 다른 모든 연산에 사용한다. 이런 사항은 gid, egid, fsgid, sgid마당에도 그룹식별자에 대해 류사하게 적용된다.

fsuid마당을 어떻게 사용하는지 보기 위해 사용자가 자신의 암호를 바꾸려는 경우

를 생각해보자. 모든 암호는 공통파일에 보관되여있지만 이 파일은 보호되여있으므로 사용자가 직접 편집할수 없다. 따라서 사용자는 /usr/bin/passwd라는 setuid기발이 1로 설정되여있으며 파일의 소유자가 초사용자인 프로그람을 호출한다. 쉘로 생성(fork)한 프로쎄스가 이와 같은 프로그람을 실행하면 프로쎄스의 euid와 fsuid 마당은 0 즉 초사용자의 PID로 설정된다. 이제 핵심부가 호출조종를 실행하면서 보면 fsuid값이 0이므로 프로쎄스는 파일에 호출할수 있다. 물론 /usr/bin/passwd 프로그람은 사용자에게 자신의 암호를 변경하는일만 허락한다.

우리는 Unix의 오랜 력사를 통해 setuid프로그람이 매우 위험하다는 사실을 알고 있다.

악의가 있는 사용자는 프로그람작성자가 전혀 계획하지 않은 연산을 setuid프로그람이 실행하도록 코드안에 있는 프로그람오유를 동작시킨다. 최악의 경우 이때문에 전체 체계의 안전이 위태로울수 있다. Linux는 이런 위험을 최소화하려고 다른 최신Unix체계와 마찬가지로 프로쎄스는 필요할 때에만 setuid특권을 얻고 더는 필요없으면 버릴수 있게 한다. 이 특성이 매우 유용하다는 사실은 여러 보호준위가 있는 사용자응용프로그람을 구현할 때 나타난다. 프로쎄스서술자에 suid마당이 있는데 이 마당에는 setuid프로그람을 실행한 직후의 유효식별자(euid와 fsuid)값이 들어간다. 프로쎄스는 setuid(), setresuid(), setfsuid(), setreuid()체계호출을 사용하여 유효식별자를 바꿀수 있다.

표 8-14에서는 이 체계호출들이 프로쎄스믿음권한에 어떻게 영향을 주는지 보여준다.

주의할점은 호출하는 프로쎄스가 초사용자권한이 없으면 즉 euid마당이 0이 아니면 프로쎄스의 믿음권한마당에 포함된 값을 설정하는 일에만 이 체계호출을 사용할수 있다.

례를 들어 일반 사용자프로쎄스가 setfsuid()체계호출을 진행하여 500이라는 값을 프로쎄스의 fsuid마당에 넣을수 있지만 다른 믿음권한마당중 하나가 이미 500이란 값을 담고있을 때에만 가능하다.

표 8-14. 프로쎄스믿음권한을 설정하는 체계호출의 의미

마당	setuid(e)		setresuid (u,e,s)	setreuid(u,e)	setfsuid(f)
	euid=0	euid≠0			
uid	e로 설정	변화없음	u로 설정	u로 설정	변화없음
euid	e로 설정	e로 설정	e로 설정	e로 설정	변화없음
fsuid	e로 설정	e로 설정	e로 설정	e로 설정	f로 설정

suid	e로 설정	변화없음	s로 설정	e로 설정	변화없음

사용자 ID마당 4개사이의 때로는 복잡한 관계를 리해하기 위해 setuid()체계호출을 생각해보자. 호출하는 프로쎄스의 euid마당이 0으로 설정되였는지(즉 프로쎄스가 초사용자권한을 소유하고있는지) 또는 일반UID로 설정되였는지에 따라 실제동작이 다르다.

euid마당이 0이면 체계호출을 진행하는 프로쎄스의 모든 믿음권한마당(uid, euid, fsuid, suid)을 파라메터 e의 값으로 설정한다. 초사용자프로쎄스는 자신의 권한을 포기하고 일반사용자가 소유한 프로쎄스가 될수 있다. 이 경우는 례를 들어 사용자가 가입할 때 발생한다. 체계는 새로운 프로쎄스를 초사용자권한으로 생성하지만 프로쎄스는 setuid()체계호출을 진행하여 자신의 권한을 포기하고 사용자의 가입쉘프로그람을 실행한다.

GID의 유효믿음권한은 대응하는 setgid(), setgid(), setfsgid(), setregid()체계호출을 사용하여 바꿀수 있다.

euid마당이 0이 아니면 체계호출은 euid와 fsuid에 보관된 값만 변경하고 다른 두 마당은 변경하지 않는다. 이것은 setuid 프로그람을 실행하는 프로쎄스가 euid와 fsuid에 보관된 유효권한을 uid(프로쎄스는 실행파일을 실행한 사용자처럼 동작한다.) 또는 suid(프로쎄스는 실행파일을 소유한 사용자처럼 동작한다.)중에서 선택하여 설정할수 있게 한다.

3. 프로쎄스의 자격

Linux는 자격(capability)개념에 기반을 둔 새로운 프로쎄스자격모형을 향해 나아가고있다. 간단히 말하면 자격은 프로쎄스가 특정클라스의 연산이나 특정연산을 수행하도록 허가 받았는지를 나타내는 기발이다. 이 모형은 euid에 따라 프로쎄스가 모든 일을 할수 있거나 아무일도 할수 없는 전통적인 《초사용자 대 일반사용자》모형과 다르다. 표 8-15에서 보여주는바와 같이 Linux핵심부는 여러 자격을 포함한다.

표 8-15. Linux자격

이　　름	설　　　　명
CAP_AUDIT_WRITE	망련결소케트에 대한 쓰기시 검사통보발생을 허가한다.
CAP_AUDIT_CONTROL	망련결소케트들에 의한 핵심부검사동작조종을 허가한다.
CAP_CHOWN	파일과 그룹소유권한의 변경에 대한 제한을 무시한다.

CAP_DAC_OVERRIDE	파일호출권한을 무시한다.
CAP_DAC_READ_SEARCH	파일/등록부읽기, 탐색권한을 무시한다.
CAP_FOWNER	파일소유에 따른 호출제한을 무시한다.
CAP_FSETID	setuid와 setgid기발설정제한을 무시한다.
CAP_KILL	신호생성할 때 권한검사를 무시한다.
CAP_IPC_LOCK	페지와 공유기억기토막에 잠그기를 허용한다.
CAP_IPC_OWNER	IPC소유권검사를 생략한다.
CAP_LEASE	파일임대(lease)를 허용한다(12장《Linux파일잠그기》참고).
CAP_LINUX_IMMUTABLE	추가전용 혹은 변경할수 없는 Ext2/Ext3파일의 변경을 허용한다.
CAP_MKNOD	mknod() 특권연산권한을 허용한다.
CAP_NET_ADMIN	일반망환경관리를 허용한다.
CAP_NET_BIND_SERVICE	1024번이하의 TCP/UDP 소케트결합(binding)을 허용한다.
CAP_NET_BROADCAST	현재 사용되지 않는다.
CAP_NET_RAW	RAW와 PACKET 소케트사용을 허용한다.
CAP_SETGID	그룹프로쎄스특질조작에 대한 제한을 무시한다.
CAP_SETRCAP	특질조작을 허용한다.
CAP_SETUID	사용자프로쎄스자격조작에 대한 제한을 무시한다.
CAP_SYS_ADMIN	일반체계관리를 허용한다.
CAP_SYS_BOOT	reboot()사용을 허용한다.
CAP_SYS_CHROOT	chroot()사용을 허용한다.
CAP_SYS_MODULE	핵심부모듈의 삽입과 삭제를 허용한다.
CAP_SYS_NICE	nice()와 setpriority()체계호출의 권한검사를 생략하고 실시간프로쎄스의 생성을 허

	용한다.
CAP_SYS_PACCT	프로쎄스계정(accounting)[a]설정을 허용한다.
CAP_SYS_PTRACE	임의의 프로쎄스에 대한 ptrace()사용을 허용한다.
CAP_SYS_RAWIO	ioperm()과 iopl()을 통한I/O포구호출을 허용한다.
CAP_SYS_RESOURCE	자원제한을 증가시키는것을 허용한다.
CAP_SYS_TIME	체계시간과 실시간시간의 변경을 허용한다.
CAP_SYS_TTY_CONFIG	말단설정을 위한 vhangup()체계호출의 실행을 허용한다.

자격의 주되는 우점은 각 프로그람이 항상 제한된수만큼의 자격만을 필요로 한다는 점이다.

따라서 악의를 품은 사용자가 오유가 존재하는 프로그람을 리용하는 방법을 발견하였다고 해도 제한된 연산류형만을 불법으로 실행할수 있다. 례를 들어 오유가 있는 프로그람에 CAP_SYS_TIME자격만 있다고 하자. 이 경우 오유를 악용할 방법을 발견한 사용자는 단지 실제시간과 체계시간만 불법으로 변경할수 있을뿐이며 다른 특권연산은 실행할수 없다.

현재VFS와 Ext2파일체계는 자격모형을 지원하지 않으므로 프로쎄스가 어떤 파일을 실행할 때 적용해야 하는 자격집합을 실행파일에 지정할 방법은 없다. 그러나 어떤 프로쎄스가 CAP_SETPCAP자격을 소유하고있다면 이 프로쎄스는 capget()과 capset()체계호출을 사용하여 명시적으로 자격을 얻거나 설정할수 있다. 례를 들어 login프로그람을 수정하여 다른 프로그람의 자격중 일부를 보유하거나 제거할수 있도록 할수 있다.

Linux핵심부는 이미 자격을 고려하고있다. 례를 들어 사용자가 프로쎄스의 정적우선순위를 변경할수 있게 하는 nice()체계호출을 생각해보자. 전통적인 모형에서는 초사용자만이 우선순위를 높일수 있다. 핵심부는 호출하는 프로쎄스서술자안에 있는 euid마당이 0으로 설정되였는지 검사해야 한다. 그러나 Linux핵심부는 CAP_SYS_NICE라는 자격을 정의하고있으며 이 자격은 정확히 이런 종류의 연산에 대응한다. 핵심부는 capable()함수에 CAP_SYS_NICE를 파라메터로 전달하여 이 기발값을 검사한다. 이 호출방법은 핵심부코드에 추가된 《호환성해킹》덕분에 동작한다.

프로쎄스가 euid와 fsuid마당은 0으로 설정할 때마다 핵심부는 모든 검사가 성공하

도록 모든 프로쎄스자격을 설정한다. 프로쎄스가 euid와 fuid마당을 프로쎄스소유자의 실제UID로 재설정하면 핵심부는 프로쎄스서술자의 keep_capabilities기발을 검사하고 기발이 설정되여있으면 모든 자격을 제거한다. 프로쎄스는 Linux고유의 prctl()체계호출을 사용하여 keep_capabilities기발을 설정하거나 지울수 있다.

4. 명령행인자와 쉘환경

사용자가 명령을 입력하면 적재된 프로그람은 요구를 만족시키기 위해 쉘에서 명령행인자를 얻는다. 례를 들어 사용자가 /usr/bin등록부파일의 전체 목록을 보려고 다음 명령을 입력하였다고 하자.

$ ls 1 /usr/bin

쉘프로쎄스는 이 명령을 실행하기 위해 새로운 프로쎄스를 생성한다. 이 새로운 프로쎄스는 /bin/ls실행파일을 적재한다. 이 과정에서 쉘에서 물려받은 실행문맥대부분을 잃게 되지만 독립된 3개 인자 ls, -1, /usr/bin은 남는다. 일반적으로 새로운 프로쎄스는 인자를 임의의 수만큼 받을수 있다.

명령행인자를 전달하는 규약은 어떤 고급언어를 사용했는가에 따라 다르다. C언어에서 프로그람의 main()함수는 프로그람에 전달된 인자가 몇개인지 나타태는 옹근수와 문자렬지적자배렬의 주소를 파라메터로 받는다. 다음은 이 표준을 형식화한것이다.

int main(int argc, char *argv[])

앞에 있는 실례로 돌아가서 /bin/ls프로그람을 호출하면 argc에는 3이 들어가고 argv[0]은 ls문자렬을 가리킨다. 또한 argv[1]은 1문자렬을 가리키고 argv[2]는 /usr/bin문자렬을 가리킨다. argv배렬의 끝은 언제나 null지적자로 표시되므로 argv[3]에는 NULL이 들어간다.

C언어에서 환경변수를 포함하고있는 파라메터를 main()함수의 세번째 파라메터로 전달할수 있다. 이 파라메터는 프로쎄스의 실행문맥를 변경하거나 사용자나 다른 프로쎄스에 정보를 전달하거나 프로쎄스가 execve()체계호출과정에서 정보를 전달하기 위해 사용할수 있다.

환경변수를 리용하려면 main()함수를 다음과 같이 선언해야 한다.

int main(int argc, char *argv[], char *envp[])

envp파라메터는 다음과 같은 형식의 환경문자렬에 대한 지적자의 배렬을 가리킨다.

VAR_NAME=something

여기서 VAR_NAME은 환경변수의 이름을 나타내고 구분자 = 뒤에 있는 문자렬은 변수에 실제로 할당된 값을 나타낸다. envp배렬의 끝은 argv배렬처럼 null지적자로 표시된다.

envp배렬의 주소는 C서고의 environ 대역변수에도 보관된다.

명령행인자와 환경변수문자렬은 사용자방식탄창에서 되돌이주소 바로 앞에 위치한

다.(《파라메터전달》참고) 사용자방식탄창의 바닥위치는 그림 8-7에서 볼수 있다.

환경변수는 0인 긴옹근수(long integer) 바로 다음에 오는 탄창의 바닥근처에 위치한다.

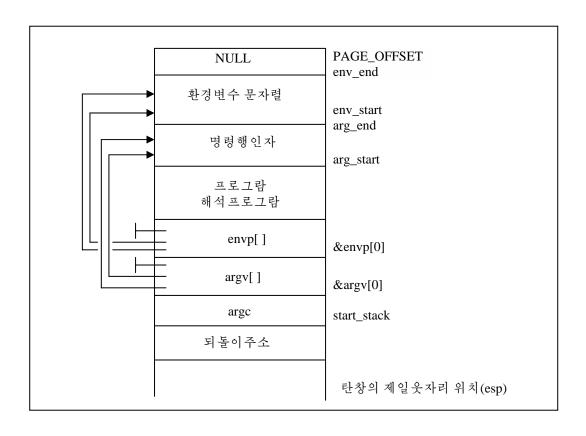



그림 8-7. 사용자방식탄창의 바닥위치

5. 서 고

각 고급언어의 원천코드파일은 여러 단계를 거쳐서 목적파일(object file)로 바뀐다. 목적파일은 고급언어명령에 대응하는 기호언어명령어코드를 포함한다. 목적파일은 원천코드파일외부의(서고나 동일한 프로그람의 다른 원천코드파일에 있는 함수 같은) 대역기호이름에 대응하는 선형주소를 포함하지 않으므로 실행할수 없다. 이와 같은 주소할당 또는 주소해석(resolution)은 련결프로그람(linker)이 수행하는데 프로그람의 모든 목적파일을 모아서 실행파일을 생성한다. 또한 련결프로그람은 프로그람에서 사용하는 서고함수를 분석하고 뒤에서 설명하는 방식으로 실행파일에 함수를 붙인다.

아주 간단한 프로그람을 포함해서 모든 프로그람이 C서고를 사용한다. 례를 들어 다음과 같이 한 행인 C프로그람을 보자.

void main(void) {}

비록 이 프로그람은 아무일도 하지 않지만 실행환경을 설정하기 위해(뒤에 나오는 《exec계렬함수》참고) 그리고 프로그람이 완료한 다음 프로세스를 제거하기 위해서 많은 작업이 필요하다.(3장에 있는 《프로쎄스끝내기》참고) 특히 C콤파일러는 main()함수완료부분의 목적코드에 exit()함수호출을 추가한다.

3절에서 프로그람이 보통 C서고에 있는 래퍼루틴을 통해 체계호출을 실행한다는 사실을 보았다. 이것은 C콤파일러의 경우도 마찬가지이다. 프로그람의 문장을 콤파일하여 생성한 코드를 포함할뿐만아니라 어떤 실행파일이든 사용자방식프로쎄스와 핵심부의 호상작용을 처리하는 《접착코드(glue code)》를 포함한다. 일부접착코드는 C서고에 보관되여있다.

Unix체계에는 C서고외에도 다른 많은 함수서고가 있다. 기본적인 Linux체계에도 서로 다른 서고가 50개이상 있다. 이중 일부만 보아도 수학서고 libm에는 실수연산을 위한 고급함수가 있으며 X11서고 libX11에는 X11윈도우체계의 도형대면부를 위한 기본적인 저수준함수를 모아놓았다.

전통적인 Unix체계의 실행파일은 정적서고(static library)를 기반으로 한다. 즉 련결프로그람이 생성한 실행파일에 원본프로그람의 코드뿐만아니라 프로그람이 참조하는 서고함수의 코드까지 들어간다는 사실을 의미한다. 정적서고의 큰 부족점은 디스크에서 많은 공간을 차지한다는 점이다. 실제로 정적으로 련결된 실행파일은 서고코드의 일부를 복제한다.

최근의 Unix체계는 공유서고(shared library)를 사용한다. 실행파일은 서고목적코드를 포함하지 않고 서고이름을 가리키는 정보만 포함한다. 프로그람을 실행하려고 기억기에 적재하면 실행파일안에 들어있는 서고이름을 분석하고 체계등록부나무에서 서고를 찾아서 요구한 코드를 실행중인 프로쎄스가 사용할수 있게 한다. 프로쎄스는 dlopen()서고함수를 리용하여 실행시간에 추가적인 공유서고를 적재할수도 있다.

공유서고는 파일기억기배치를 제공하는 체계에서는 프로그람을 실행하는데 필요한 주기억기의 량을 감소시키므로 특히 유용하다. 프로그람해석프로그람이 어떤 공유서고를 프로쎄스에 련결해야 할 때 목적코드를 복사하지 않고 단지 서고파일의 관련부분을 프로쎄스의 주소공간에 기억기배치한다. 이렇게 함으로써 서고의 기계어코드를 포함하는 페지들을 같은 코드를 사용하는 모든 프로쎄스사이에서 공유한다.

공유서고 역시 부족점은 있다. 대개 동적으로 련결된 프로그람을 시작하는데 걸리는 시간이 정적으로 련결된 프로그람의 시작시간에 비해 훨씬 길다. 그리고 동적으로 련결된 프로그람은 같은 서고의 다른 판본을 포함한 체계에서 제대로 동작하지 않을수도 있으므로 정적련결보다 이식성이 떨어진다.

사용자는 언제나 프로그람을 정적으로 련결하도록 요구할수 있다. 례를 들어 GCC 콤파일러는 static선택항목을 제공하여 련결프로그람이 공유서고대신 정적서고를 사용

하게 할수 있다.

6. 프로그람토막과 프로쎄스기억기령역

Unix프로그람의 선형주소공간은(론리적인 관점으로 볼 때) 이전부터 토막이라는 몇개의 선형주소구간으로 나뉘어있었다.

본문토막

실행코드를 포함한다.

초기화된 자료토막

초기화된 자료를 포함한다. 즉 정적변수와 대역변수의 초기값을 실행파일에 보관한다.(프로그람은 시작할 때 해당 값을 알아야 하기때문이다.)

초기화되지 않은 자료토막

초기화되지 않은 자료를 포함한다. 즉 실행파일에 초기화값을 보관하지 않은 모든 대역변수를 보관한다.(프로그람이 참조하기 전에 값을 설정해야 하기때문이다.) 력사적인 리유로 bss토막라고도 부른다.

탄창토막

반환주소, 파라메터, 실행하는 함수의 국부변수 등을 담은 프로그람탄창을 포함한다.

각 mm_struct기억기서술자는 (3장의 《기억기서술자》참고) 프로쎄스의 특정기억기령역의 역할을 나타내는 다음과 같은 마당을 포함한다.

start_code, end_code

프로그람원본코드 즉 실행파일의 코드를 포함하는 기억기령역의 시작과 끝 선형주소를 보관한다. 본문토막은 공유서고를 포함하지만 실행가능한 파일을 포함하지 않으므로 이 마당이 나타내는 기억기령역은 본문토막의 부분집합하다.

start_data, end_data

실행파일에 지정된것처럼 프로그람원본자료를 포함하는 기억기령역의 시작과 끝 선형주소를 보관한다. 이 마당은 자료토막에 대응하는 기억기령역을 대략적으로 나타낸다. 실제로 start_data는 언제나 end_code 바로 다음에 오는 첫번째 페지주소가 되므로 이 마당을 사용하지는 않지만 end_data마당은 사용한다.

start_brk, brk

동적으로 할당된 프로쎄스의 기억기령역을 포함한 기억기령역의 시작과 끝 선형주소를 보관한다.(3장의 《동적기억관리》참고) 이 기억기령역을 동적기억기(heap)라고 부르기도 한다.

start_stack

main()의 되돌이주소 바로 우에 있는 주소를 보관한다. 그림 8-7을 보면 웃자리주소는 예약되여있다.(탄창은 아래방향으로 증가한다.)

arg_start, arg_end

명령행인자를 포함하는 탄창부분의 시작과 끝 주소를 보관한다.

env_start, env_end

환경변수문자렬을 포함하는 탄창부분의 시작과 끝 주소를 보관한다.

각 공유서고는 우의 목록과 다른 기억기령역에 배치되므로 공유서고와 파일기억기배치는 프로그람토막에 기초한 프로세스의 주소공간구분을 무의미하게 만드는 측면이 있다.

이제 간단한 례를 사용하여 Linux핵심부가 공유서고를 프로세스의 주소공간에 어떻게 배치하는지 살펴보자. 사용자방식주소공간을 0x00000000-0xbfffffff까지 가정한다.

조작체계의 바깥계층을 구현하는 모든 프로세스를 생성하고 활동을 감시하는(3장의 《핵심부스레드》참고) /sbin/init프로그람을 살펴보자. 표 8-16에서는 init프로세스에 대응하는 기억기령역을 보여준다.(이 정보는 /proc/1/maps파일에서 얻은것이다. 물론 init프로그람의 판본과 콤파일되고 련결된 방식에 따라 아래표와 다른 표를 볼수도 있다 .) 렬거된 모든 령역은 비공개(private)기억기배치를 통해 구현되였다.(표 8-13에서 권한렬에 있는 p문자) 이것은 놀랄 일이 아니다. 이 기억기령역들은 단지 프로세스에 자료를 제공하기 위해 존재한다. 프로세스는 명령을 실행하는 도중에 이 기억기령역들의 내용을 수정하지만 이와 관련한 디스크우에 있는 파일은 바뀌지 않고 남아있다. 이것은 비공개기억기배치의 동작방식과 일치한다.

표 8-16. init프로세스의 기억기령역

주소범위	권한	배치된 파일
0x08048000-0x0804cfff	r-xp	/sbin/init의 편위0
0x0804d000-0x0804cfff	rw-p	/sbin/init의 편위0x4000
0x0804e000-0x0804efff	rwxp	배치된 파일 없음
0x40000000-0x40014fff	r-xp	/lib/ld-2.2.3.so의 편위0
0x40015000-0x40015fff	rw-p	/lib/ld-2.2.3.so의 편위0x14000
0x40016000-0x40016fff	rw-p	배치된 파일 없음
0x40020000-0x40126f	r-xp	/lib/libc.so.2.2.3의 편위0

ff		
0x40127000-0x4012cfff	rw-p	/lib/libc.so.2.2.3의 편위 0x10600
0x4012d000-0x40130fff	rw-p	배치된 파일 없음
0xbfffd000-0xbfffffff	rwxp	배치된 파일 없음

0x8048000에서 시작하는 기억기령역은 /sbin/init 파일의 0-20479B범위에 해당하는 기억기배치이다.(/proc/1/maps파일에서는 령역의 시작과 끝만 얻을수 있지만 이로부터 령역의 크기를 쉽게 계산할수 있다.) 령역에 지정된 권한은 실행가능하고(목적코드를 포함하고있음) 읽기전용이며(명령은 실행중에 바뀌지 않으므로 쓸수 없음) 비공개이다. 따라서 이 령역이 프로그람의 본문토막를 배치함을 추측할수 있다.

0x804d000에서 시작하는 기억기령역은 /sbin/init파일의 16384(표 8-13의 0x4000) - 20479B범위에 해당하는 기억기배치이다. 이 령역에 쓰기권한이 지정되여있으므로 프로그람의 자료토막를 배치한다고 추측할수 있다.

다음의 0x804e000부터 시작하는 한 페지크기 기억기령역은 anonymous로서 어떤 파일과도 련관되지 않으며 init의 bss토막에 배치된다. 이와 비슷하게 0x40000000, 0x40015000, 0x40016000부터 시작하는 다음의 세 기억기령역은 각각 /lib/ld.2.2.3.so서고의 본문토막, 자료토막, bss토막에 대응한다.

실제로 이 서고는 ELF공유서고에 대한 프로그람해석프로그람이다. 프로그람해석프로그람은 절대 단독으로 실행되지 않으며 언제나 다른 프로그람을 실행하는 프로쎄스의 주소공간내부에 기억기배치된다. 이 체계에서 C서고는 /lib/libc.2.2.3.so파일에 보관되여있다. C서고의 본문토막, 자료토막, bss토막은 0x40020000에서 시작하는 다음 세 기억기령역에 배치된다. 비공개령역에 포함된 페지들은 《쓰기복사》기구를 사용하여 바뀌지 않으면 여러 프로쎄스가 공유할수 있다. 본문토막은 읽기전용이므로 현재 실행중인 대부분의 프로쎄스는 C서고의 실행코드를 포함하는 페지들을 공유한다.(정적으로 련결된 실행파일 제외)

끝으로 0xbfffd000 - 0xbfffffff범위에 있는 마지막 anonymous기억기령역은 사용자방식탄창과 련관된다. 3장의 《페지절환례외조종기》에서 설명한것처럼 탄창은 필요할 때마다 자동적으로 낮은 주소방향으로 증가한다.

7. 실행추적

실행추적(execution tracing)은 프로그람이 다른 프로그람의 실행을 감시하도록하는 기술이다. 추적당하는 프로그람은 신호를 받을 때까지 한 단계씩 단계별로 실행되거나 혹은 체계호출을 실행할 때까지 실행될수 있다. 실행추적은 중단점(breakpoint)

삽입과 실행중 변수에 호출하는 기술과 함께 오유수정기에서 주로 사용한다. 여기서는 오유수정기의 동작이 아닌 핵심부가 실행추적을 어떻게 지원하는지에 초점을 맞춘다.

Linux에서는 ptrace()체계호출을 통해 실행추적을 실행한다. 이 체계호출은 표 8-14에 있는 명령어를 처리할수 있다. CAP_SYS_PTRACE자격기발이 설정된 프로쎄스는 체계에서 init이외의 모든 프로쎄스를 추적할수 있다. 반대로 CAP_SYS_PTRACE 자격이 없는 프로쎄스 P는 P와 소유자가 같은 프로쎄스만 추적할수 있다. 그리고 동시에 두 프로쎄스가 한 프로쎄스를 추적할수 없다.

표 8-17. ptrace 명령어

명령어	설명
PTRACE_TRACEME	현재프로쎄스에 대해 실행추적을 시작한다.
PTRACE_PEEKTEXT	본문토막에서 32bit 값을 읽는다.
PTRACE_PEEKDATA	자료토막에서 32bit 값을 읽는다.
PTRACE_PEEKUSR	CPU의 일반, 오유수정등록기를 읽는다.
PTRACE_POKETEXT	본문토막에 32bit 값을 쓴다.
PTRACE_OLDSETOPTIONS	PTRACE_SETOPTIONS에 동등한 구성방식종속명령
PTRACE_POKEDATA	자료토막에 32bit 값을 쓴다.
PTRACE_POKEUSR	CPU의 일반, 오유수정등록기를 쓴다.
PTRACE_CONT	실행을 계속한다.
PTRACE_KILL	추적당한 프로쎄스를 완료한다.
PTRACE_GETSIGINFO	추적당한 프로쎄스에 넘겨준 마지막신호에 대한 정보를 얻는다.
PTRACE_SETSIGINFO	추적당한 프로쎄스에 넘겨준 마지막신호에 대한 정보를 꾸며낸다.
PTRACE_SINGLESTEP	단일 기호언어명령어에 대해 계속 실행한다.
PTRACE_GETREGS	CPU의 특권등록기를 읽는다.
PRRACE_SETREGS	CPU의 특권등록기를 쓴다.
PTRACE_GETFPREGS	실수등록기를 읽는다.
PTRACE_SETFPREGS	실수등록기를 쓴다.
PTRACE_GETFPXREGS	MMX와XMM등록기를 읽는다.
PTRACE_SETFPXREGS	MMX와XMM등록기를 쓴다.
PTRACE_ATTACH	다른 프로쎄스에 대해 실행추적을 시작한다.
PTRACE_DETACH	실행추적을 완료한다.

PTRACE_GET_THREAD_AREA	추적당한 프로세스의 리익(behalf)에 대한 스레드국부저장(TLS)령역을 얻는다.
PTRACE_SET_THREAD_AREA	추적당한 프로세스의 리익에 대한 스레드국부저장령역을 설정한다.
PTRACE_GETEVENTMSG	추적당한 프로세스로부터 추가자료를 얻는다. (례: 새로 호출된 프로세스의 PID)
PTRACE_SETOPTIONS	ptrace()동작을 수정한다.
PTRACE_SYSALL	다음체계호출경계까지 실행을 계속한다.

ptrace()체계호출은 추적당한 프로세스의 서술자내에 있는 p_pptr마당을 수정하여 추적하는 프로세스를 가리키도록 한다. 그 결과 추적하는 프로세스는 추적당하는 프로세스의 실제적인 부모가 된다. 실행추적을 완료하면 즉 PTRACE_DETACH 명령을 주어 ptrace()를 호출하면 체계호출은 p_pptr을 p_opptr값으로 설정한다. 즉 추적당한 프로세스의 원래부모로 설정한다.(3장의 《프로세스사이 친족관계》참고)

추적당한 프로그람에 여러 감시사건을 련관지을수 있다.

・한 기호언어명령어의 실행이 끝남

・체계호출에 들어감

・체계호출에서 빠져나옴

・신호를 수신함

감시중인 사건이 발생하면 추적하는 프로그람을 중단하고 SIGCHLD신호를 부모에 전달한다. 부모프로세스가 자식을 계속 실행하고싶으면 감시하려는 사건종류에 따라 PTRACE_CONT, PTRACE_SINGLESTEP, PTRACE_SYSCALL명령어중 하나를 사용할수 있다.

PTRACE_CONT명령어는 실행을 재개한다. 자식은 다른 신호를 받을 때까지 실행을 계속한다.

이 종류의 추적은 프로세스서술자의 PF_PTRACED기발로 구현하며 do_signal()함수가 검사한다.(3장 3절의 《신호받기》참고)

PTRACE_SINGLESTEP명령어는 자식프로세스가 다음기호언어명령어를 실행하고 난 뒤 다시 중단하게 한다. 이 종류의 추적은 Intel x86기반기계에서 eflags등록기의 TF함정기발을 사용하여 구현한다. 이 기발을 1로 설정하면 《오유수정》레외가 모든 기호언어명령 다음에 발생한다. 대응하는 레외조종기는 기발을 해제하고 현재프로세스를 중단한다. 그리고 부모에 SIGCHLD신호를 전송한다. TF기발을 설정하는 연산은 특권을 소유한 연산이 아니며 사용자방식프로세스는 ptrace()체계호출 없이도 프로그람을 한 단계씩 실행할수 있다. 핵심부는 프로세스서술자의 PF_DTRACE기발을 검사하여 자식프로세스가 ptrace()를 통해 한 단계씩 실행되는지 추적한다.

PTRACE_SYSCALL명령은 추적당하는 프로세스가 체계호출을 진행할 때까지 계속 실행하게 한다. 프로세스는 두번 중단된다. 첫번째는 체계호출을 시작할 때 두번째는 체계호출을 완료할 때이다. 이런 추적은 프로세스서술자내의 PF_TRACESYS기발로 구현하며 기호언어로 구현한 system_call()함수가 검사한다.(3절의 《system_call()함수》참고)

Intel펜티움처리기의 오유수정자격을 사용하여 프로세스를 추적할수도 있다. 례를 들어 부모프로세스가 PTRACE_POKEUSR명령을 사용하여 자식프로세스의 dr0, … dr7 오유수정등록기값을 설정할수 있다. 수신된 사건이 발생하면 CPU는 오유수정례외를 발생시킨다.

례외조종기는 추적당한 프로세스를 보류하고 SIGCHLD신호를 부모프로세스에 보낸다.

8. 실행파일형식

공식적인 Linux의 실행파일형식은 ELF(Executalbe and Linking Format)이다. 이 형식은 Unix체계연구소(USL)에서 개발했으며 Unix계렬에서 가장 많이 사용한다. System V Release 4(SVR4)와 SUN의 Solaris 2 등 잘 알려진 Unix조작체계에서 ELF를 기본실행파일형식으로 사용한다.

이전Linux판본에서는 a.out(Assembler OUTput Format)라는 다른 형식을 지원하였다. 실제로 Unix계렬에는 a.out형식이 여러 종류가 있다. 그러나 ELF가 훨씬 더 실용적이므로 요즘은 이 형식을 잘 사용하지 않는다.

Linux는 다른 여러 형식의 실행파일을 지원한다. 이를 통해 MS-DOS의 EXE프로그람이나 Unix BSD의 COFF실행파일과 같이 다른 조작체계용으로 콤파일한 프로그람을 실행할수 있다. 그러나 bash스크립트와 같은 몇가지 실행파일형식은 기반에 독립적이다.

실행파일의 형식은 linux_binfmt류형의 객체로 나타내는데 핵심적으로 3가지 메쏘드를 제공한다.

load_binary

실행파일에 보관된 정보를 읽어서 현재프로세스를 위한 새로운 실행환경을 설정한다.

load_shlib

이미 실행중인 프로세스에 공유서고를 동적으로 결합하는데 사용한다. uselib()체계호출을 통해 이루어진다.

core_dump

현재프로세스의 실행문맥를 core라는 파일에 보관한다. 이 파일의 형식은 실행중인 프로그람의 실행류형에 따라 다른데 프로세스가 기본동작이 《dump》로 설정된 신호를 받으면 이 파일을 생성한다.(3장 《신호를 받았을 때의 동작》참고)

모든 linux_binfmt객체는 단방향련결목록에 포함되고 목록에서 첫번째 요소의 주소는 formats변수에 보관된다. register_binfmt()와 unregister_binfmt()함수를 호출하여 목록에 요소를 삽입하거나 제거할수 있다. 핵심부에 콤파일되여있는 각 실행파일형식에 대해서는 체계시작과정에 register_binfmt()함수를 실행한다. 이 함수는 새로운 실행파일형식을 구현하는 모듈을 적재할 때 사용하며 모듈을 부리울 때에는 unregister_binfmt()함수를 실행한다.

formats목록의 마지막항목은 언제나 해석된 스크립트(interpreted script)를 위해 실행가능한 파일형식을 설명하는 객체이다. 이 형식은 load_binary메쏘드만 정의한다. 대응하는 do_load_script()함수는 실행파일이 #!라는 두 문자로 시작하는지 검사한다. 만일 그렇다면 첫번째 행의 나머지를 다른 실행가능한 파일의 경로명으로 해석하고 파라메터에 스크립트파일명을 전달하여 실행하려 한다. 스크립트 파일이 #!문자로 시작하지 않더라도 사용자의 쉘이 인식할수 있는 언어로 작성한 파일의 경우에는 실행할수 있다. 이 경우 사용자가 명령을 입력하는 쉘에서 스크립트를 해석하며 핵심부는 직접 관여하지 않는다.

Linux는 사용자가 독자적인 실행파일형식을 등록할수 있게 한다. 파일의 처음 128B에 보관된 식별번호나 파일류형을 식별하는 파일확장자를 사용하여 각 형식을 알아낸다. 례를 들어 MS-DOS의 확장자는 파일명에서 점(.)으로 구분되는 세 문자로 이루어진다.

.exe확장자는 실행프로그람을 나타내고 .bat확장자는 쉘 스크립트를 나타낸다.

각각의 독자적인 형식에 해석프로그람이 련관되며 핵심부는 원래의 독자적인 실행파일명을 파라메터로 넘겨 이 해석프로그람을 자동적으로 실행한다. 이 기구는 스크립트방식과 류사하지만 독자적인 형식에 아무런 제한을 하지 않으므로 더욱 강력하다.

새로운 형식을 등록하려면 사용자는 /proc/sys/fs/binfmt_misc

/register파일에 다음과 같은 형식의 문자렬을 쓴다.

:name:type:offset:string:mask:interpreter:

각 마당의 의미는 다음과 같다.

name

새로운 형식에 대한 식별자

type

인식류형(M은 식별번호, E는 확장자)

offset

파일내부식별번호의 시작편위

string

식별번호안이나 확장자안에서 일치하는 바이트렬

mask

string 안에 있는 일부비트를 제거(mask out)하기 위한 문자렬

interpreter

프로그람해석기의 전체 경로명

flags

프로그람해석기가 의존해야 할 방식을 조종하는 몇가지항목기발

례를 들어 초사용자가 다음 명령을 실행하면 핵심부가 MS Windows의 실행파일형식을 인식할수 있게 한다.

```
$ echo ':DOSWin:M:O:MZ:0xff:/usr/local/bin/wine:' >\
              /proc/sys/fs/binfmt_misc/register
```

Windows실행파일은 처음 두 바이트에 MZ라는 식별번호를 가지고 /usr/local/bin/wine이라는 프로그람해석기가 실행한다.

9. 실행도메인

Linux의 훌륭한 자격으로 다른 조작체계용으로 콤파일한 파일을 실행할수 있는 능력이 있다. 물론 이것은 파일에 핵심부가 실행중인것과 같은 콤퓨터기본방식에 대한 기계어코드가 들어있을 때에만 가능하다. 이와 같은 《외래(foreign)》프로그람에 대해 다음과 같은 두가지를 지원한다.

・모방실행 : POSIX를 준수하지 않는 체계호출을 포함한 프로그람을 실행하기 위해 사용한다.

・직접실행 : 모든 체계호출이 POSIX를 준수하는 프로그람일 경우에 사용할수 있다.

마이크로소프트의 MS-DOS와 Windows 프로그람은 모방실행방식으로 처리된다. 이 프로그람은 Linux가 인식하지 못하는 API를 포함하므로 직접 실행할수 없다. 그대신 DOSemu 또는 Wine과 같은 모방기를 호출하여 각 API호출을 모방하는 래퍼함수호출로 변환하고 함수호출은 기존의 Linux체계호출을 사용한다. 모방기는 대부분 사용자방식응용프로그람으로 구현하므로 더 다루지 않는다.

반면에 Linux이외의 조작체계에서 콤파일한 POSIX준수프로그람은 POSIX조작체계들이 류사한 API를 제공하기때문에 큰 문제없이 실행할수 있다.(실제로 API가 동일해야 하지만 항상 그런것은 아니다.) 핵심부가 조정해야 하는 작은 차이들은 체계호출을 어떻게 호출하는지, 여러 신호에 어떤 번호가 붙어있는지 등이다. 이 정보들은 exec_domain형 실행도메인서술자(exection domain descriptor)에 보관된다.

프로쎄스가 자신의 실행도메인을 지정하기 위해서는 서술자의 personality마당을 설정하고 exec_domain마당안에 대응하는 exec_domain자료구조의 주소를 보관한다.

프로쎄스는 personality()라는 체계호출을 실행하여 자신의 특성을 변경할수 있다.

체계호출의 파라메터에 사용하는 전형적인 값은 표 8-15와 같다. 프로그람작성자가 직접 자신의 프로그람특성을 바꾸리라 생각하지 않으므로 C서고에는 대응하는 래퍼

루틴이 없다.

대신 프로세스의 실행문맥를 설정하는 접착코드에서 personality()체계호출을 실행해야 한다.(《exec계렬함수》참고)

표 8-18. Linux핵심부가 지원하는 주요특성

특 성	조 작 체 계
PER_LINUX	표준실행도메인
PER_LINUX_32BIT	64bit 구성방식에서 32bit 물리주소를 가지고있는 Linux
PER_LINUX_FDPIC	ELF FDPIC형식에서 Linux프로그람
PER_SVR4	System V Release4
PER_SVR3	System V Release3
PER_SCOSVR3	SCO Unix판본 3.2
PER_OSR5	SCO Open Server Release5
PER_WYSEV386	Unix System V/386 Release 3.2.1
PER_ISCR4	Interactive Unix
PER_BSD	BSD Unix
PER_SUNOS	Sun OS
PER_XENIX	Xenix
PER_LINUX32	64bit 구성방식에서 Linux 32bit 프로그람의 모의(4GB사용자방식주소공간을 사용)
PER_LINUX32_3GB	64bit 구성방식에서 32bit 프로그람들의 모의(3 GB 사용자방식주소공간을 사용)
PER_IRIX32	32bit SGI Irix-5
PER_IRIXN32	32bit SGI Irix-6
PER_IRIX64	64bit SGI Irix-6
PER_RISCOS	RISC OS
PER_SOLARIS	Sun Solaris
PER_UW7	Caldera Unixware 7
PER_OSF4	Digital UNIX (Compaq Tru64 UNIX)
PER_HPUX	Hewlett-Packard화사의 HP-UX

10. exec계렬함수

Unix체계는 프로세스의 실행문맥를 실행파일이 나타내는 새로운 문맥으로 교체하

는 여러 함수를 제공한다. 각 함수의 이름은 exec로 시작하고 뒤에 한두 글자가 더 붙는다. 따라서 이런 부류의 함수를 《exec계렬함수》라고 부른다.

표 8-19에서 exec계렬함수를 보여준다. 이것들의 차이점은 파라메터를 해석하는 방법이다.

표 8-19. exec 계렬함수

함수명	PATH탐색	명령행인자	환경변수배렬
execl()	No	목록	No
execlp()	Yes	목록	No
execle()	No	목록	Yes
execv()	No	배렬	No
execvp()	Yes	배렬	No
execve()	No	배렬	Yes

각 함수의 첫번째 인자는 실행할 파일의 경로명을 나타낸다. 경로명은 절대경로일수도 있고 프로세의 현재등록부에서 상대경로일수도 있다. 그리고 이름에 /문자가 없으면 execlp()와 execvp()함수는 실행가능한 파일을 PATH환경변수가 지정하는 모든 등록부에서 찾는다.

첫번째 파라메터외에 execl(), execlp(), execle()함수는 다양한 추가파라메터를 포함한다. 각각 새로운 프로그람을 위한 명령행인자를 나타내는 문자렬을 가리킨다. 함수명에서 문자 1이 의미하듯이 파라메터는 NULL값으로 끝나는 목록에 정리되여있다.

보통 첫번째 명령행인자는 실행가능한 파일명을 복제한다. 반면에 execv(), execvp(), execve()함수는 명령행인자를 파라메터 하나로 나타낸다. 함수명의 문자 v가 의미하듯이 파라메터는 명령행인자문자렬을 가리키는 지적자의 벡토르주소이다. 배렬의 마지막요소에는 NULL값을 보관한다.

execle()와 execve()함수는 마지막파라메터로 환경변수문자렬을 가리키는 지적자배렬의 주소를 받는다. 배렬의 마지막요소에는 NULL값을 넣어야 한다. 다른 함수는 environ외부대역변수로부터 새로운 프로그람을 위한 환경변수를 리용할수 있다. 이 변수는 C서고에 정의되여있다.

execve()를 제외한 모든 exec계렬함수는 execve()를 사용하여 C서고에 정의된 래퍼루틴이다. execve()는 프로그람실행을 위해 Linux에서 제공하는 유일한 체계호출이다.

sys_execve()봉사루틴은 다음과 같은 파라메터를 받는다.

- 실행파일의 경로명의 주소(사용자방식주소공간에 있음)

・문자렬(사용자방식주소공간)을 가리키는 지적자로 구성된 배렬(사용자방식주소공간)의 주소배렬은 NULL로 끝난다. 각 문자렬은 명령행인자를 나타낸다.

・문자렬(사용자방식주소공간)을 가리키는 지적자로 구성된 배렬(사용자방식주소공간)의 주소배렬은 NULL로 끝난다. 각 문자렬은 NAME=value형식의 환경변수를 나타낸다.

함수는 실행파일의 경로명을 새로 할당한 페지틀에 복사한다. 그리고 do_execve() 함수를 호출하면서 페지틀, 지적자 배렬, 사용자방식등록기내용을 보관한 핵심부방식탄창의 위치 등을 가리키는 지적자를 파라메터로 전달한다. do_execve()는 다음과 같은 연산을 수행한다.

1. linux_binprm자료구조를 정적으로 할당한다. 여기에는 새로운 실행파일에 관련된 자료가 들어갈것이다.

2. path_init(), path_walk와 dentry_open()을 호출하여 덴트리객체, 파일객체, 실행파일에 대응하는 색인마디객체를 얻는다. 실패하면 적절한 오유코드를 반환한다.

3. 색인마디의 i_writecount마당을 검사하여 실행가능한 파일이 기록중인지 확인한다. 나중에 쓰기호출을 방지하기 위해 이 마당을 1로 보관한다.

4. prepate_binprm()함수를 호출하여 linux_binprm자료구조를 채운다. 이 함수는 다음과 같은 연산을 수행한다.

a. 파일의 호출권한이 실행을 허용하는지 검사한다. 허용하지 않으면 오유코드를 반환한다.

b. 실행파일의 setuid와 setgid기발값을 고려하여 linux_bin_prm구조체의 e_uid와 e_gid마당을 설정한다. 이 마당은 각각 유효한 사용자ID와 유효한 그룹ID를 나타낸다. 또한 프로쎄스자격을 검사한다.(앞서 살펴본 《프로쎄스믿음권한과 자격》에서 호환성해킹을 설명하였다.)

c. linux_binprm구조체의 buf마당을 실행파일의 처음 128B로 채운다. 이 바이트에는 실행파일형식의 식별번호와 실행파일을 인식하기 위한 적절한 정보들이 들어있다.

5. 파일의 경로명, 명령행인자, 환경변수 문자렬 등을 새로 할당된 페지틀에 복사한다.(결국 이것들은 사용자방식주소공간에 배치될것이다.)

6. search_binary_handler()함수를 호출한다. 이 함수는 formats목록을 탐색하고 linux_binprm 자료구조를 파라메터로 하여 각 항목의 load_binary메쏘드를 호출한다. formats목록의 탐색은 load_binary메쏘드가 파일의 실행파일형식을 인식하는데 성공하면 즉시 완료한다.

7. 실행파일형식이 formats목록에 없으면 할당된 모든 페지틀을 해제하고 ENOEXEC오유코드를 반환한다. Linux는 실행파일형식을 인식할수 없다.

8. 그렇지 않으면 파일의 실행파일형식에 대응하는 load_binary메쏘드에서 얻은 코드를 반환한다.

실행파일형식에 대응하는 load_binary메쏘드는 다음과 같은 연산을 실행한다.(실행파일이 파일기억기배치를 허가하는 파일체계에 보관되여있으며 공유서고를 하나 이상 요구한다고 가정한다.)

1. 파일의 처음 128B에 보관된 식별번호를 검사하여 실행파일형식을 식별한다. 식별번호가 일치하지 않으면 NOEXEC오유코드를 반환한다.

2. 실행파일의 머리부를 읽는다. 이 머리부는 프로그람의 토막과 필요한 공유서고를 설명한다.

3. 실행파일에서 프로그람해석기의 경로명을 얻는다. 공유서고의 위치를 찾아 기억기에 배치하기 위해 이 경로명을 사용한다.

4. 프로그람해석기의 덴트리객체(그리고 색인마디객체와 파일객체)를 얻는다.

5. 프로그람해석기의 실행권한을 검사한다.

6. 프로그람해석기의 처음 128B를 완충기에 복사한다.

7. 프로그람해석기의 류형에 대해 일관성검사를 수행한다.

8. flush_old_exec()함수를 호출하여 이전계산에서 사용한 자원의 대부분을 해제한다. 이 함수는 다음과 같은 연산을 수행한다.

ㄱ. 신호조종기의 표를 다른 프로쎄스와 공유하고있으면 새로운 표를 할당하고 이전표의 사용계수기를 감소시킨다. make_private_signals()함수를 호출하여 수행한다.

ㄴ. exec_mmap()함수를 호출하여 기억기서술자, 모든 기억기령역과 프로쎄스와 프로쎄스의 페지표에 할당한 모든 페지들을 해제한다.

ㄷ. 신호조종기표의 각 신호를 기본동작으로 다시 설정한다. release_old_sighals()와 flush_signal_handlers()함수를 호출하여 수행한다.

ㄹ. 프로쎄스서술자의 comm마당을 실행파일경로이름으로 설정한다.

ㅁ. flush_thread()함수를 호출하여 실수등록기값과 TSS토막에 보관된 오유수정등록기를 지운다.

ㅂ. de_thread()를 호출하여 프로쎄스를 이전스레드그룹에서 분리한다.(3장의 《프로쎄스식별》참고)

ㅅ. flush_old_files()함수를 호출하여 프로쎄스서술자의 files->close_on_exec마당에 대응하는 기발이 설정된 모든 열린 파일을 닫는다.(2장의 《프로쎄스관련파일》 참고)

이제 되돌아갈수 없는 지점에 이르렀다. 무엇인가 잘못되였더라도 함수를 이전으로 되돌릴수 없다.

9. 프로쎄스의 새로운 특성 즉 프로쎄스서술자의 personality마당을 설정한다.

10. 프로쎄스서술자의 PF_FORKNOEXEC기발을 지운다. 새로운 프로쎄스가 생성(fork)되면 설정되고 이 프로쎄스가 새로운 프로그람을 실행하면 지워지는 이 기발은

프로쎄스계정을 위해 필요하다.

11. setup_arg_pages()함수를 호출하여 프로쎄스의 사용자방식탄창을 위한 새로운 기억기령역서술자를 할당하고 해당 기억기령역을 프로쎄스의 주소공간으로 추가한다. setup_arg_pages()는 명령행인자와 환경변수문자렬을 포함하는 페지들을 새로운 기억기령역에 배치한다.

12. do_mmap()함수를 호출하여 실행가능한 파일의 본문토막(즉 코드)을 배치하는 새로운 기억기령역을 생성한다. 일반적으로 실행가능한 코드는 재배치할수 없으므로 기억기령역의 시작선형주소는 실행가능한 파일형식에 의존한다. 따라서 함수는 본문토막이 특정한 론리주소편위에서(특정선형주소에서) 시작하여 적재될것이라고 가정한다. ELF프로그람은 선형주소 0x8048000에서 시작하는 선형주소에 적재된다.

13. do_mmap()함수를 호출하여 실행파일의 자료토막를 배치하는 새로운 기억기령역을 생성한다. 그런데 실행가능한 코드 역시 자신의 변수를 특정편위(즉 특정선형주소)에서 찾을수 있기를 기대하므로 기억기령역의 시작선형주소는 실행파일형식에 의존한다. ELF프로그람의 경우 자료토막은 본문토막 바로 다음에 적재된다.

14. 실행가능파일의 또 다른 특수한 토막를 위해 추가기억기령역을 할당한다. 일반적으로는 아무것도 없다.

15. 프로그람해석기를 적재하는 함수를 호출한다. 프로그람해석기가 ELF실행파일이면 함수는 load_elf_interp()이다. 이 함수는 보통 11에서 13단계까지 연산을 수행하지만 파일이 아닌 프로그람해석기를 실행한다. 프로그람해석기의 본문과 자료를 포함하는 기억기령역의 시작주소는 프로그람자신이 지정한다. 그러나 그 주소는 실행할 파일의 본문과 자료를 배치할 기억기령역과의 충돌을 피하려고 매우 높은장소(보통 0x40000000)에 위치한다.(앞서 살펴본 《프로그람토막과 프로쎄스기억기령역》참고)

16. 파일서술자의 binfmt마당에 실행파일형식의 linux_binfmt객체의 주소를 보관한다.

17. 프로쎄스의 새로운 자격을 결정한다.

18. 프로그람해석기표를 생성하여 명령행인자와 환경변수문자렬의 지적자배렬사이에 있는 사용자방식탄창에 보관한다.(그림 8-7참고)

19. 프로쎄스의 기억기서술자의 start_code, end_code, end_data, start_brk, start_stack 마당을 설정한다.

20. do_brk()함수를 호출하여 프로그람의 bss토막을 배치하는 새로운 anonymous기억기령역을 생성한다.(프로쎄스가 변수에 기록하면 요구페지화을 개시하고 페지틀을 할당한다.) 실행프로그람이 련결될 때 이 기억기령역의 크기를 계산한다. 일반적으로 프로그람의 실행가능코드는 재배치할수 없으므로 기억기령역의 초기선형주소를 반드시 명시해야 한다. ELF프로그람에서는 bss토막이 data토막 바로 다음에 적재된다.

21. start_thread()마크로를 호출하여 핵심부방식탄창에 보관된 사용자방식등록기 eip와 esp값을 변경하여 각각 프로그람해석기의 진입지점과 새로운 사용자방식탄창의 맨우를 가리키도록 한다.

22. 프로쎄스가 추적당하고있으면 프로쎄스에 SIGTRAP신호를 보낸다.

23. 0을 반환한다.(성공)

execve()체계호출을 마치고 호출한 프로쎄스가 사용자방식에서 실행을 계속하게 되면 실행문맥이 크게 바뀐다. 체계호출을 진행한 코드는 더는 존재하지 않는다.

이런 의미에서 execve()는 절대성공하면서 되돌이하지 못한다고 할수 있다.

대신 새로운 프로그람을 프로쎄스의 주소공간에 배치하여 실행한다.

그러나 프로그람해석기는 공유서고의 적재을 처리해야 하므로 아직 새로운 프로그람을 실행할수 없다. 실행파일이 정적으로 련결되여 공유서고를 요구하지 않으면 아주 간단하다. load_binary 메쏘드는 프로그람의 text, data, bss, stack 토막을 프로쎄스기억기령역에 배치하고 사용자방식 eip등록기를 새로운 프로그람의 진입지점으로 설정한다.

비록 프로그람해석기가 사용자방식에서 실행하지만 어떻게 동작하는지 간단히 살펴보자. 첫번째 작업은 핵심부가 환경변수문자렬과 arg_start를 가리키는 지적자배렬사이에 있는 사용자방식탄창에 보관한 정보에서 시작하여 자신을 위한 기본적인 실행문맥을 설정한다. 그리고 프로그람해석기는 실행될 프로그람을 검사하여 어떤 공유서고를 적재할것인지, 각 공유서고안에 어떤 함수를 요구할것인지 결정한다. 그리고 해석프로그람은 mmap()체계호출을 진행하여 실제로 프로그람이 사용한 서고함수를 보관할 페지를 배치하는 기억기령역을 생성한다. 다음으로 해석프로그람은 서고의 기억기령역의 선형주소에 따라 공유서고의 기호에 대한 모든 참조를 갱신한다. 마지막으로 프로그람해석기는 실행된 프로그람의 기본진입점으로 이동하면서 자신의 실행을 완료한다. 이제부터 프로쎄스는 실행파일과 공유서고코드를 실행한다.

여기서 살펴본바와 같이 프로그람을 실행하는 일은 프로쎄스추상화, 기억기관리, 체계호출, 파일체계 같은 다양한 핵심부설계와 관련한 복잡한 작업이다.

Linux핵심부해설서

집 필	리수철, 리영훈 한명일, 김만준	심 사	박종혁
편 집	김철우	교 정	박석채
장 정	서경애	콤퓨터편성	여은정
낸 곳	교육성 교육정보쎈터	인쇄소	교육성 교육정보쎈터
인 쇄	주체97(2008)년 8월 10일	발 행	주체97(2008)년 8월 20일

교-07-1314